Hes·so VALAIS WALLIS
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences Western Switzerland

Studiengang Systemtechnik

Vertiefungsrichtung Infotronics

Diplom 2009

Thomas Nanzer

Prozessorsystem auf FPGA

Dozent	François Corthay
Experte	Prof. Ivan Defilippis

Sitten, 6. July 2009

HES-SO Valais

SI	TV	EE	IG	EST
X	X			

Données du travail de diplôme
Daten der Diplomarbeit

FO 1.2.02.07.DB
mam/30/06/2008

☒ FSI ☐ FTV	Année académique / *Studienjahr* **2008/09**	No TD / *Nr. DA* **it/2009/20**
Mandant / *Auftraggeber* ☒ HES—SO Valais ☐ Industrie ☐ Etablissement partenaire	Etudiant / *Student* **Thomas Nanzer**	Lieu d'exécution / *Ausführungsort* ☒ HES—SO Valais ☐ Industrie ☐ Etablissement partenaire
Professeur / *Dozent* **François Corthay**	Expert / *Experte* *(données complètes)*	
Travail confidentiel / *vertrauliche Arbeit* ☐ oui / *ja* ☒ non / *nein*		

Titre / *Titel*

Prozessorsystem auf FPGA

Description et Objectifs / *Beschreibung und Ziele*

Die HES-SO//Wallis hat ein neues FPGA-Board erstellt. Das Board sollte genug Ressourcen haben, um ein Mikroprozessorsystem zu enthalten: FPGA, Flash, RAM, Ethernet, Slave USB, JTAG und RS-232.

Ziel dieser Arbeit ist es, ein Prozessorsystem aufzustellen:

— Leon- und AMBA-System mit Hilfe von AMBArchitect erstellen
— Mikroprozessorsystem in der FPGA prototypieren und mit Hilfe der Debug Support Unit (DSU) überprüfen
— Auf dem PC einen Monitor zum Debug-Interface und zum Programmieren der Flash schreiben
— Ethernet und/oder USB testen.

Signature ou visa / *Unterschrift oder Visum* Resp. de la filière *Leiter des Studieng.:* ..... Etudiant / *Student:*	Délais / *Termine* Attribution du thème / *Ausgabe des Auftrags:* 18.02.2009 Remise du rapport / *Abgabe des Schlussberichts:* 06.07.2009, 12:00 Exposition publique / *Ausstellung Diplomarbeiten:* 04.09.2009 Défense orale / *Mündliche Verfechtung:* Semaine / *Woche* 35

Rapport reçu le / *Schlussbericht erhalten am* ..... Visa du secrétariat / *Visum des Sekretariats* .....

Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences Western Switzerland

Prozessorsystem auf FPGA

Diplomand/in Thomas Nanzer

Ziel des Projekts

Auf Basis des neuen FPGA Embedded System Board ein Prozessorsystem inkl. Schnittstellencontroller und Speicherkontroller für FLASH und SDRAM entwickeln und konfigurieren.

Methoden / Experimente / Resultate

Das Prozessorsystem verwendet von Aeroflex Gaisler entwickelte IP Cores. Herzstück ist der LEON3 Mikroprozessor. Ebenfalls werden Schnittstellencontroller Cores eingesetzt, diese umfassen z.B. Ethernet und RS-232. Als Hauptkommunikationsmittel wird der AMBA Bus eingesetzt.
Hauptaufgabe ist die Konfiguration dieser Cores damit eine Implementierung auf dem FPGA_EBS_V2.0 Board möglich wird.
Auf Grund von technischen Limitierungen musste während der Diplomarbeit auch ein System zur Programmation des FLASH Speichers entwickelt werden. Ebenfalls konnte nicht wie geplant ein komplettes System-on-a-chip entwickelt werden, da die verwendete FPGA nicht genügend Kapazität aufweist. So wurde das System auf eine Minimalkonfiguration ohne Ethernet und USB getrimmt.
Die entwickelten Systeme und Schaltungen funktionieren sowohl in der Simulation als auch in der Praxis auf der Hardware.

Diplomarbeit
| 2009 |

Studiengang
Systemtechnik

Anwendungsbereich
Infotronics

Verantwortliche/r Dozent/in
François Corthay
Francois.Corthay@hevs.ch

Partner
HES-SO Valais

HES-SO Wallis
Route du Rawyl 47
1950 Sitten

Tel. 027 606 85 11
Web www.hevs.ch

Prozessorsystem auf FPGA

—

Technische Dokumentation

Diplomarbeit 2009

Autor:	Nanzer Thomas
Projekt:	Prozessorsystem auf FPGA
Experten:	François Corthay, Prof. Ivan Defilippis
Start:	11. Mai 2009
Ende:	06. Juli 2009
Version:	1.0

Inhaltsverzeichnis

Abbildungsverzeichnis

Algorithmenverzeichnis

Tabellenverzeichnis

1 Einleitung

Nach Abschluss des Systemtechnik-Infotronics Bachelor Studiengangs an der HES-SO Wallis wird während 8 Wochen von jedem Student eine individuelle Diplomarbeit abgelegt. Die Projekte werden zum Grossteil von den Dozenten der Schule vorgeschlagen und anschliessend zugeteilt. Während dem Frühlingssemester wurde den Studenten ein Tag pro Woche als Einarbeitungszeit hin zur Diplomarbeit zur Verfügung gestellt.

In dieser Diplomarbeit wird ein Mikroprozessorsystem in einem Field Programmable Gate Array (FPGA) realisiert. Die FPGA ist auf einer neuen HES-SO Wallis Platine verbaut. Neben dem Prozessor werden auch diverse Schnittstellen wie USB oder RS-232, welche sich auch auf der Platine befinden, in Betrieb genommen.

Dieses Dokument beschreibt das Projekt in allgemeiner und technischer Hinsicht. Der Umfang reicht über die Planung, Implementierung bis hin zu den abschliessenden Tests. Es dient als Leitfaden durch das Projekt und ermöglicht ein erneutes Realisieren einer solchen Aufgabe oder ist die Basis für zukünftige Aufgaben.

1.1 FPGA

Die Abkürzung FPGA steht für Field Programmable Gate Array und ist ein programmierbarer Integrierter Schaltkreis (Integrated Circuit, IC) in der Digitaltechnik. Wie der Name schon sagt, lässt sich der interne Aufbau des Chips flexibel, sogar "vor Ort", modifizieren. Es können einfache aber auch hoch-komplexe Schaltungen mit FPGAs realisiert werden und dies, vor allem bei Kleinserien, sehr günstig. Sie werden vor allem dann angewendet wenn es auf schnelle Signalverarbeitung ankommt oder um flexible Änderungen einer Schaltung sicherzustellen.

In einer FPGA kann beispielsweise ein "Soft"-Prozessor implementiert werden. Dem zur Seite können diverse Schnittstellen, Controller, Speicher, Konverter usw. stehen. Alle Komponenten innerhalb der FPGA können parallel arbeiten. Ein solcher flexibler Aufbau wird intern durch programmierbare logische Komponenten und einem ebenfalls programmierbaren Verbindungsnetz mit steuerbaren Schaltern realisiert. Damit die FPGA mit der Aussenwelt kommunizieren kann, verfügt sie ebenfalls über I/O-Blöcke. Diese werden mit Pins an die Schaltmatrix angebunden. Auch für die Taktaufbereitung sind spezielle Blöcke, z.b. PLLs, in modernen FPGAs untergebracht. Manche FPGAs bieten ebenfalls fest verdrahteten Block RAM an.

Programmiert werden solche Chips mit einer Hardwarebeschreibungssprache. VHDL und Verilog werden oft eingesetzt. Very High Speed Integrated Circuit Hardware Description Language, kurz VHDL, wurde in den 1980er Jahren entworfen und das US Verteidigungsministerium verhalf der Sprache zum Durchbruch. VHDL ermöglicht eine schnelle Entwicklung von selbst grossen und komplexen Schaltungen. Hervorzuheben ist vor allem die Tatsache, dass VHDL Instruktionen parallel ablaufen. So können selbst sehr zeitkritische Aufgaben gelöst werden. Ebenfalls besteht die Möglichkeit Libraries anzulegen oder einzusetzen. Alternativ gibt es auch Entwicklungsumgebungen, welche den grafischen Entwurf der FPGA ermöglichen. Die Konfiguration wird in Form einer Schaltung mit Blöcken und Verbindungen gezeichnet. Daraus lässt sich dann Code in einer Hardwarebeschreibungssprache generieren.
Die Konfiguration wird meist in externen, nicht flüchtigen Speicherelementen (FLASH, EEPROM) abgelegt und kann beim Start des Systems in die FPGA geladen werden.

1.2 Aeroflex Gaisler & LEON

Aeroflex Gaisler[1] wurde von Jiri Gaisler gegründet und ist ein Unternehmen, welches sich auf Digitales Hardware Design spezialisiert hat. Sie bieten zahlreiche IP (Intellectual Property) Cores und Entwicklungshilfsmittel an, vieles davon unter der GNU General Public License.
Herzstück ist der LEON3 Prozessor und die GRLIB. Der LEON3 ist ein 32-Bit Prozessor basierend auf der SPARC V8 Architektur. Seine Vorgänger wurden für die European Space Agency (ESA) entwickelt. In der neusten Version verfügt der LEON über eine tiefere Pipeline und Multi-Prozessor Support. Die GRLIB ist eine Library bestehend aus zahlreichen IP Cores. Diese Cores können wiederverwendet werden und sind optimiert für System-On-Chip (SoC) Lösungen. Die Cores werden über einen Bus miteinander verbunden und verfügen über eine von Gaisler entwickelte Plug & Play Methode, welche die Zusammenarbeit der IP Cores beinahe automatisiert. In der GRLIB sind, neben dem LEON3, beispielsweise ein Speichercontroller, Ethernetcontroller, CAN Controller und viele weitere Cores zu finden.

[1] http://www.gaisler.com/

1.3 AMBA Bus & AMBAdraw

Advanced Microcontroller Bus Architecture, oder kurz AMBA, wurde von ARM Limited entwickelt und definiert drei Busse. Advanced High-Performance Bus (AHB), Advanced System Bus (ASB) und Advanced Peripheral Bus (APB). AMBA wird in der Industrie und auch in der Infotronics Abteilung der HES-SO Wallis oft eingesetzt. Mit Hilfe des AMBA Busses werden Komponenten in SoC Systemen verbunden. Zudem lässt sich dieses Bussystem, typischerweise aus einem AHB und APB inkl. Bridge bestehend, sehr gut in einer FPGA realisieren. Die bereits erwähnte GRLIB von Gaisler basiert auf dem AMBA Bus.

AMBAdraw[2] ist ein Entwicklungshilfsmittel, welches an der HES-SO Wallis entwickelt wurde. Die Software ermöglicht das Einfügen und Konfigurieren der GRLIB IP Cores, deren Platzierung auf dem AMBA Bus und ein Export als VHDL Code oder als Mentor HDL Designer Projekt. Die einfache Bedienung und die umfangreichen Funktionalitäten zeichnen diese unter der GPL Lizenz veröffentlichte Software aus.

2 Ziel

2.1 Situation

An der HES-SO Wallis wurde eine neue Platine entworfen, FPGA_EBS_V2.0. Das Board wird im Rahmen dieser Diplomarbeit verwendet und bietet eine Vielzahl an Schnittstellen an. Darunter USB, Ethernet, RS-232, ein General Purpose Interface mit zwei Ports. Ebenfalls besitzt die Platine Mezzanine Schnittstellen zur Zusammenarbeit mit anderen Steckkarten oder Mikrocontrollern. Es ist auch genug Speicher vorhanden um Applikationen implementieren zu können. Dank den Fähigkeiten der verbauten FPGA sind die Möglichkeiten dieses Board sehr vielseitig.
Während diesem Projekt wird ein LEON3 System inkl. diversen Schnittstellencontrollern entwickelt. Verbunden werden die Komponenten intern mit dem AMBA Bus. Das System setzt auf die GRLIB von Aeroflex Gaisler auf. Ebenfalls wird das Entwicklungstool AMBAdraw einem Praxistest unterzogen.
Des Weiteren wird ein Debug Monitor entwickelt, dieser erlaubt es das LEON3 System während dessen Laufzeit zu stoppen und die Applikation zu debuggen indem der Speicherinhalt kontrolliert werden kann. Zu Test und Präsentationszwecken wird anschliessend eine Demoapplikation entwickelt. Die Demo ist eine Morse Code Ethernet Applikation, wobei das FPGA Board als Empfangs- und Sendestation im Einsatz ist.

[2] http://ambadraw.ch.vu/

2.2 Pflichtenheft

Ein Pflichtenheft wurde im Vorfeld definiert. Während dem sechsten Semester wurde nach einer Analyse der Aufgabenstellung ein detailliertes Pflichtenheft erstellt. Dieses beschreibt die Projektziele und gibt einen Überblick über die Funktionalitäten des Projektes wieder.
Das Pflichtenheft ist in Beilage 1 angefügt.

3 Material

In diesem Kapitel wird das eingesetzte Material beschrieben.
Bemerkung: Während des Projekts wird keine Hardware entwickelt, alles ist bereits zur Verfügung gestellt worden.

3.1 Hardware

Dieser Abschnitt gibt einen Überblick über die eingesetzte Hardware. Wie bereits erwähnt wird das neue FPGA_EBS_V2.0 Board verwendet.

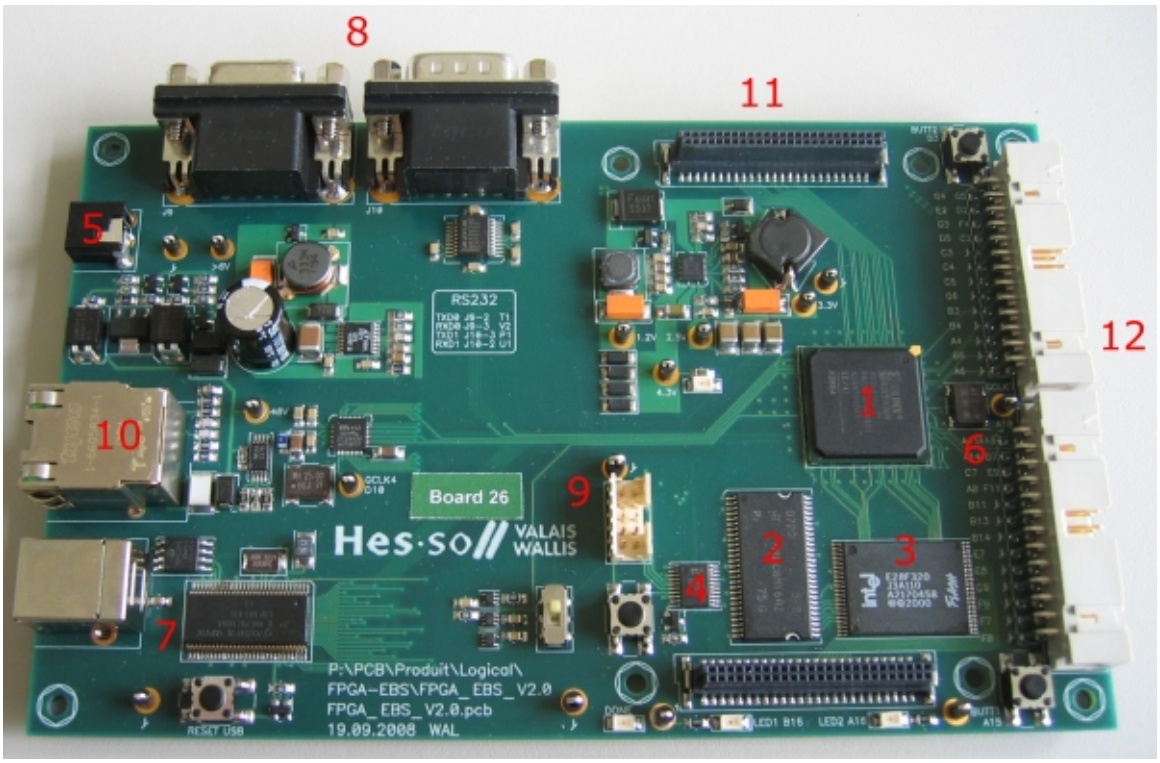


Abbildung 1: FPGA_EBS_V2.0 Board

Komponente	Name	Beschreibung
1	Xilinx FPGA Spartan-3E	XC3S500E, 500K Gates, 360K Block RAM Bits
2	Micron SDRAM MT48LC8M16A2	2Meg Addressbits x 16 Databits x 4 Banks = 16MByte
3	Intel FLASH 28F320J3A	32 x 128 KByte Blocks = 4Mbyte
4	Xilinx EEPROM XCF04S	EEPROM für FPGA Konfiguration
5	Spannungsversorgung	Board über Netzstecker, USB oder Ethernet betreibbar
6	Crystal Oscillator	Für FPGA, 66MHz
7	Cypress EZ-USB FX2 Controller	USB 2.0 Controller mit programmierbarem 8051 Prozessor, separatem EEPROM, 24MHz Quarz, USB 2.0 Type B Port
8	RS-232 Interface	1x male, 1x female Port, für Debug und allg. Kommunikation
9	JTAG Interface	Zur Programmation der FPGA
10	Micrel KSZ8041NL	10/100MBit/s Ethernet PHY Layer Controller, 25MHz Quarz, Ethernet Port
11	Mezza Interface	Zum Anschluss von Tochterkarten, 37 & 39 I/Os
12	General Purpose Interface	2x 17 Point-to-Point I/Os

Tabelle 1: FPGA_EBS_V2.0 Komponenten

Des Weiteren sind einige Schalter, Taster und LEDs auf dem Board verbaut.

3.2 Software

Folgende Software wurde während der Diplomarbeit eingesetzt. Viele der Applikationen wurden bereits während dem Studiengang kennen gelernt.

FPGA-Entwicklungssoftware

- **AMBAdraw:** v3.0.2, CAD AMBA und GRLIB Entwicklung
- **Mentor HDL Designer:** v2007.1a, CAD FPGA Entwicklung
- **Mentor ModelSim:** v6.3g, Simulationsapplikation
- **Xilinx ISE:** v10.1, Syntheseprogramm, Place and Route für FPGA Produkte von Xilinx
- **Synplicity Synplify Pro:** v9.4, Syntheseprogramm

Applikations-Entwicklungssoftware

- **GRTools:** v20081001, Software Paket, u.a. inkl. Eclipse, Eclipse LEON IDE, GRMON, usw.
- **GRMON:** v1.1.35 (eval), Debug-Software von Gaisler für LEON Prozessoren
- **BCC:** Cross-Compiler, kompatibel mit LEON Prozessoren
- **Active Perl:** v5.10, umfangreiche Perl Distribution mit vielen Libraries
- **Komodo Edit:** v5.0.3, Perl Entwicklungssoftware / Editor

Tools & Utilities

- **UltraEdit:** v10.20c, umfangreicher Texteditor, kompatibel mit vielen Programmiersprachen
- **Cygwin:** Windows kompatible UNIX Shell
- **SuPTerminal:** v1.1, Terminalsoftware für RS-232 (COM) Schnittstelle
- **Tera Term:** v3.1.3, Terminalsoftware für RS-232 (COM) Schnittstelle

4 Projektstudie/Spezifikation

In diesem Kapitel wird das Projekt genauer definiert und vorgestellt. Während der Planungsphase wird die Aufgabe in einfachere Unterblöcke geteilt und diese genau analysiert. Folgende Unterkapitel besprechen jeweils einen solchen Unterblock und spezifizieren dessen Inhalt und Funktionalität.

4.1 LEON System

Der LEON Prozessor ist das Herz des Systems. Damit jedoch der LEON Prozessor in der FPGA mit seiner Aussenwelt kommunizieren kann, braucht es zahlreiche Schnittstellen und Bussysteme. Die Implementation von diesen ist von Aeroflex Gaisler bereits erstellt worden. Gaisler bietet eine Vielfalt von IP Cores, meist unter GPL Lizenz, an. Diese können anschliessend in ein System integriert und personalisiert werden.

Die untenstehende Abbildung gibt einen Überblick über die verwendeten IP Cores von Gaisler und wie diese am AMBA Bus angeschlossen sind.

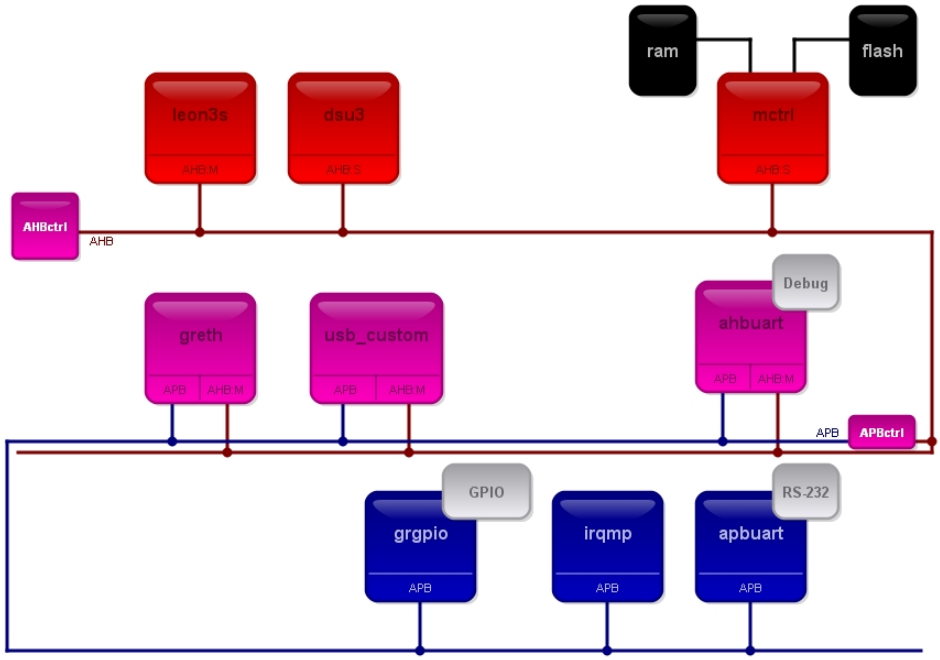


Abbildung 2: LEON System am AMBA Bus

In den folgenden Abschnitten werden alle eingesetzten Gaisler Cores vorgestellt. Die genaue Konfiguration der einzelnen Cores ist in Kapitel Realisierung zu sehen.

4.1.1 AMBA Bus & GRLIB

Wie bereits erwähnt, handelt es sich bei der GRLIB um eine Sammlung von wiederverwendbaren IP Cores, unterteilt in mehrere VHDL Libraries. Diese von Gaisler erstellte Library setzt auf den AMBA AHB und APB Bus auf und entspricht dem AMBA 2.0 Standard. Allerdings hat Gaisler den Standard durch zusätzliche Signale erweitert. Diese Zusatzfunktionen sind automatische Adressen Decodierung, Interruptsteuerung und Geräte Identifikation (Plug & Play).

4.1.1.1 AMBA AHB on-chip Bus Der AMBA Advanced High-Performance Bus ist ein Multi-Master Bus der schnelle und bandbreitenhungrige Cores miteinander verbindet. Der Bus ist ein multiplexter Bus, dies ist auch in untenstehender Abbildung zu sehen. Dabei können Master und Slave Cores angeschlossen sein. Welche Komponente zu welchem Zeitpunkt den Bus benützen darf, wird vom Bus Controller (Arbiter) bestimmt.

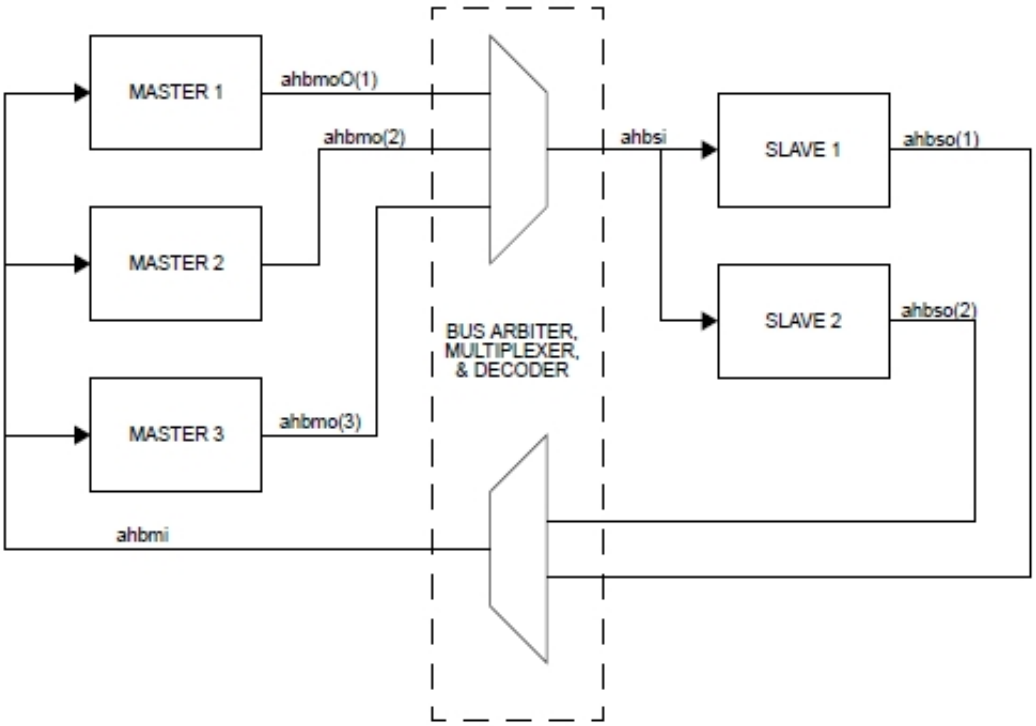


Abbildung 3: AHB Bus Aufbau

Die Signale sind in vier wichtige VHDL Records aufgeteilt:

- AHB Master Input (ahbmi)
- AHB Master Output (ahbmo)
- AHB Slave Input (ahbsi)
- AHB Slave Output (ahbso)

Der Ausgang des aktiven Masters (ahbmo) wird vom Bus Arbiter ausgewählt und anschliessend an alle Slave Inputs (ahbsi) weitergeleitet. Umgekehrt wird der Ausgang des aktiven Slaves (ahbso) vom Arbiter an alle Master Eingänge (ahbmi) weitergeleitet. Der Arbiter, welcher Adressdecoder und Bus Multiplexer ist, kontrolliert welcher Master und welcher Slave gerade aktiv sind.

In Beilage 2 ist die komplette Liste der oben erwähnten Records zu sehen. An dieser Stelle sollen nur drei Signale kurz erwähnt werden:

- hgrand: Buszuweisungssignal für Master (Eingang), bestimmt nächsten Bus Master
- hsel: Buszuweisungssignal für Slaves (Ausgang), bestimmt ausgewählten Slave
- hindex: Buszuweisungsbestätigungssignal für Master und Slaves (Ausgang), Kontrollmittel um zu bestimmen ob korrekter Core den Bus verwendet

Die AHB Plug & Play Konfiguration erlaubt es angeschlossene Cores zu identifizieren, das Adressenmapping für Slaves zu vereinfachen und Interrupts zu steuern. Jede AHB Einheit weist einen 8x 32-Bit Konfigurationsrecord auf. Dieser ist in folgender Abbildung zu sehen:

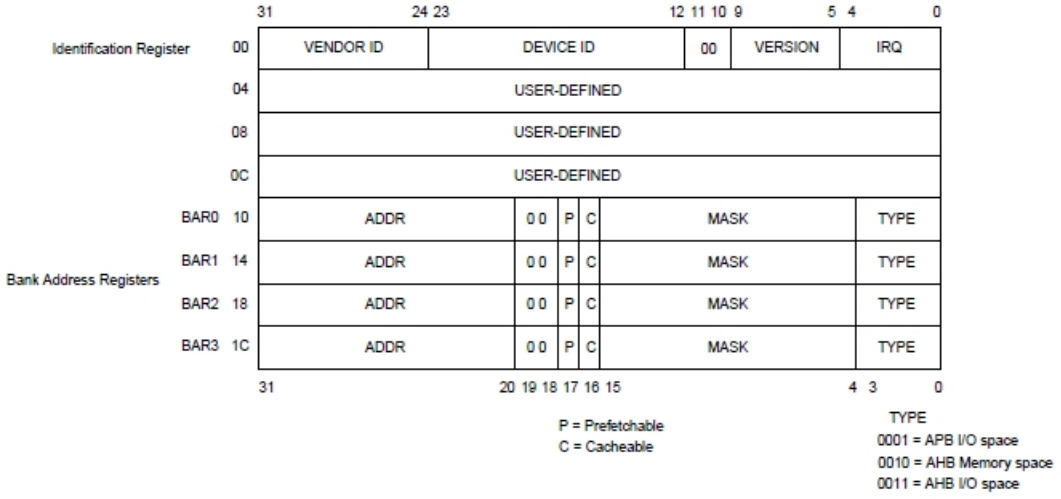


Abbildung 4: AHB Plug & Play Record

Das erste 32-Bit Word ist das Identification Register. Es umfasst Informationen zum Hersteller des Cores (Vendor ID), die Core Identifikation an sich (Device ID) sowie Interrupt Steuerung (IRQ). Die vier letzten 32-Bit Words sind Bank Address Register und enthalten Address Mapping Informationen für AHB Slaves.

Alle AHB Plug & Play Records werden in einer read-only Tabelle an einer bestimmten AHB Adresse gespeichert. Typischerweise bei 0xFFFFF000, wobei die Master Records von 0xFFFFF000 bis 0xFFFFF800 reichen und die Slave Records von 0xFFFFF800 bis 0xFFFFFFFC. Die Tabelle bietet somit für je 64 Master und Slaves Platz. Ein Plug & Play Betriebssystem oder sonst eine Applikation kann nun diese Tabelle durchlaufen und alle angeschlossenen Cores erkennen, inkl. deren Konfiguration. Die Konfigurationsrecords werden von den angeschlossenen Cores automatisch an den AHB Bus Controller via hconfig Signal verschickt. Dieser kann daraufhin die Tabelle füllen und auch immer aktuell halten.

4.1.1.2 AMBA APB on-chip Bus

Der sogenannte AMBA Advanced Peripheral Bus ist ein Single-Master Bus für einfachere, weniger komplexe Cores, welche auch kleinere Datenraten verwenden. Der Master ist zugleich die AHB/APB Brücke, welche beide Busse miteinander verbindet. Es wäre möglich, mehrere APB Busse an einen AHB Bus anzuschliessen.

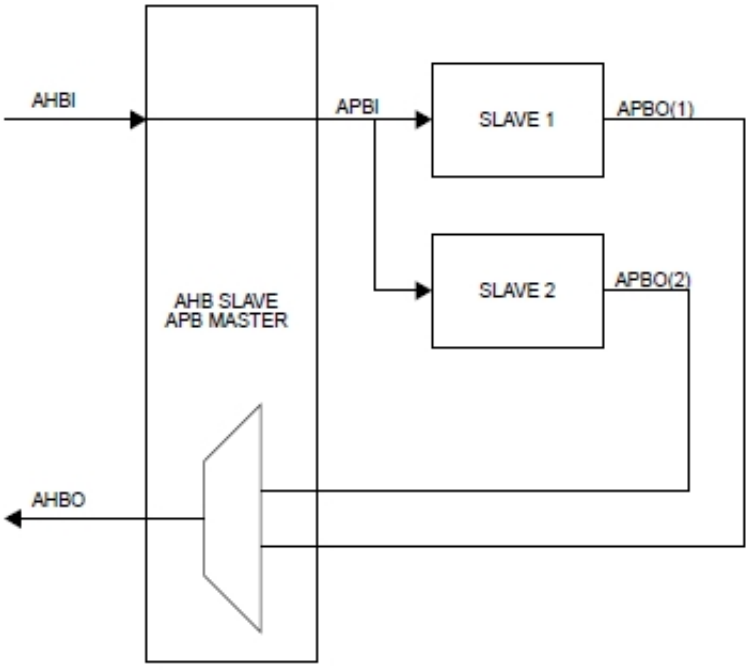


Abbildung 5: APB Bus Aufbau

Obenstehende Abbildung zeigt, dass auch dieser Bus multiplexiert ist. Zwei wichtige VHDL Records fassen die Slave Signale zusammen:

- APB Slave Input (apbi)
- APB Slave Output (apbo)

Das AHBI Signal stammt vom schnellen AHB Bus und dient der Brücke als Eingangssignal. Daraufhin wird vom APB Master das APB Eingangssignal (APBI) an alle Slaves weiter geschickt. Anschliessend wird das APB Ausgangssignal des aktiven Slaves gewählt und durch die Brüche, via AHBO, an den AHB Bus geschickt.

Ebenfalls in Beilage 2 sind detaillierte Informationen über den Aufbau der Signalrecords zu sehen. Analog zum AHB Bus gibt es auch hier ein Select Eingangssignal (psel) und ein Index Ausgangssignal (pindex).

Auch die Plug & Play Fähigkeiten des APB Busses gleichen denen des AHB Busses. Der Konfigurations Record eines APB Slaves besteht aus zwei 32-Bit Words. Dies ist in folgender Abbildung zu sehen:

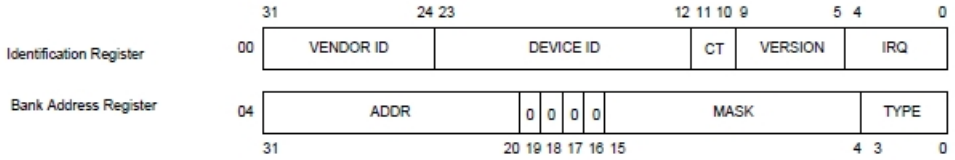


Abbildung 6: APB Plug & Play Record

Das erste Word ist auch hier das Identification Register und enthält Informationen zum Hersteller (Vendor ID) zum Core (Device ID) und zum Interrupt Routing. Das zweite Word ist das Bank Address Register und enthält Address Mapping Informationen des APB Slaves.

Die APB Adressierung wird an den AHB Adressbereich angepasst. Die 12 werthöchsten Bits des AHB Busses sind für die Adressierung der AHB/APB Brücke reserviert. Somit sind die 20 wertniedrigeren Bits für die Adressierung der APB Slaves zuständig.

Die Konfigurationsrecords werden in einer read-only Tabelle an einer fixen AHB Adresse zusammengefasst. In der Regel ab 0x—FF000 bis 0x—FFF00. Somit ist Platz für bis zu 512 Slaves an einem einzigen APB Bus. Das Erstellen und Scannen dieser Tabelle funktioniert wie beim AHB Bus.

Weitere Informationen zum AMBA Bus und dessen Implementierung in der GRLIB sind in Beilage 2 zu finden.

4.1.2 LEON3 IP Core

LEON3 ist ein 32-Bit Prozessor Core, welcher der SPARC V8 Architektur entspricht. Entwickelt wurde dieser Prozessor speziell für Embedded Systems. Dementsprechend ist es ein sehr sparsamer und kein hoch komplexer Prozessor. Jedoch ist seine Leistung mehr als ausreichend für Embedded Systems Applikationen. Er besitzt eine 7-Stage Integer Pipeline und verwendet die Harvard Architektur, d.h. der Prozessor verwaltet intern die Daten und die Instruktionen getrennt. Extern wird aber ein gemeinsamer Speicher verwendet. Dieser Architektur entsprechend lassen sich auch zwei getrennte Daten- und Befehlscaches aktivieren. Der LEON3 ist modular aufgebaut, es lassen sich intern zusätzliche Blöcke aktivieren. Beispielsweise kann eine Memory Management Unit (MMU), eine Floating Point Unit (FPU) oder auch eine Hardware Multiplikations- und Divisionseinheit zugeschaltet werden. Ebenfalls besteht die Möglichkeit eine externe Debug- sowie eine externe Interruptkomponente anzuschliessen. Angeschlossen wird der Prozessor als Master an den schnellen AMBA AHB Bus.
Folgende Abbildung zeigt das Blockschema des Prozessors auf.

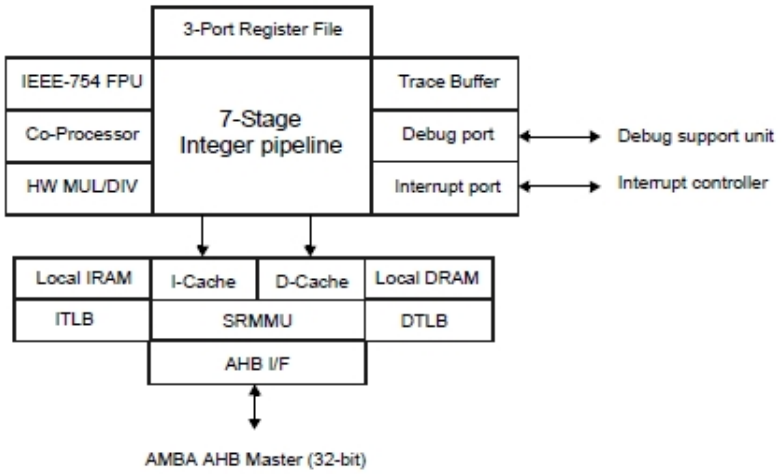


Abbildung 7: LEON3 Blockschema

Die Implementierung von Gaisler ist hoch konfigurierbar. Die einzelnen Blöcke sind nicht nur separat aktivierbar sondern lassen sich auch vielfältig einstellen. Beispielsweise kann die Anzahl der Register, von den normal acht Registern, verändert werden oder es ist möglich die Grösse und die Arbeitsweise der Cachespeicher anzupassen. Die Parameter, welche für diese Konfigurationen nötig sind, werden in der Regel mit VHDL Generics gesetzt. Während der Laufzeit können viele Einstellungen auch über Konfigurationsregister vorgenommen werden. Die 7-Stage Integer Pipeline ist in folgende Abschnitte unterteilt:

1. Instruction Fetch: der nächste Befehl wird im Cache oder Speicher gesucht.
2. Decode: der Befehl wird nun decodiert und Call/Branch Adressen vorbereitet.
3. Register Access: Operanden werden von den Registern gelesen.
4. Execute: ALU, logische und Shift Operationen werden durchgeführt.
5. Memory: Daten werden in den Daten Cache gespeichert.
6. Exception: Interrupts werden bearbeitet.
7. Write: Resultate der Execute oder Memory Stage werden zurück in die Register geschrieben.

Der LEON Prozessor ist zudem auch Synchronous Multi-Processing (SMP) fähig, bis zu 16 LEON3 Prozessoren können am gleichen AMBA Bus angeschlossen werden. Jeder erhält dann seine eigene ID. Der Prozessor kann zudem 16 verschiedene Interrupts erkennen und bietet Power-Down Stromsparfunktionen an, in welchen die Pipeline gestoppt wird bis ein nächster Interrupt eintrifft. Bei einem Reset startet der Prozessor bei Adresse 0x0.

Eine detailliertere Beschreibung sowie eine Auflistung aller Interrupts, Generics und Signale ist im GRLIB Auszug in Beilage 3 zu sehen.

4.1.3 AHBCTRL IP Core

Dieser Core ist ein AHB Arbiter, Bus Multiplexer und Slave Decoder gemäss AMBA 2.0 Standard in einem Block vereint. Es können bis zu 16 AHB Master und 16 AHB Slaves angeschlossen werden.
Die Arbitration kann mit fixer Priorität oder im abwechselnden Round-Robin Verfahren erfolgen. Im Round-Robin Modus lassen sich zusätzlich priorisierte Master definieren. Nach jedem AHB Transfer wechselt das Zugriffsrecht auf dem Bus. Bei Burst Zugriffen bleibt der Bus dem Master zugeteilt bis der Burst Zugriff beendet ist. Das Slave Decoding wird mit Hilfe der Plug & Play Fähigkeiten der Gaisler Cores realisiert.

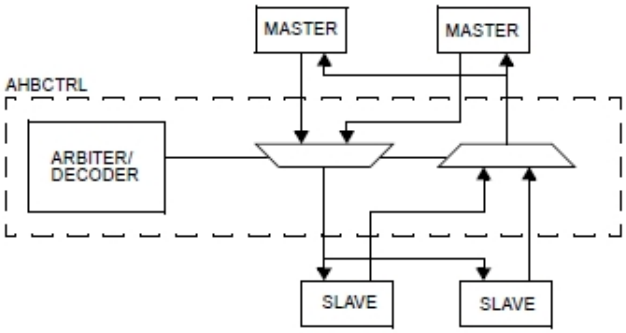


Abbildung 8: AHBCTRL Blockschema

Konfiguriert wird der Core wiederum über VHDL Generics. Ein Auszug der GRLIB ist in Beilage 4 angefügt.

4.1.4 APBCTRL IP Core

Dieser Block stellt die Brücke zwischen dem schnellen AHB und dem langsameren APB Bus dar. Er ist der einzige Master des APB Busses. Bis zu 16 Slaves können angeschlossen werden. Das Decoding der Slaves wird wiederum mit der Plug & Play Methode von Gaisler durchgeführt.

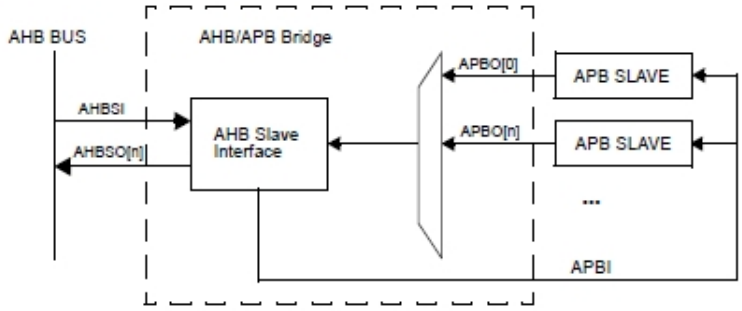


Abbildung 9: APBCTRL Blockschema

In Beilage 5 ist der GRLIB Auszug beigelegt. Darin sind auch alle Generics und Signale dieses Cores aufgeführt.

4.1.5 DSU3 IP Core

Der LEON3 Prozessor bietet einen Debug Modus an, in welchem die Pipeline angehalten wird und der Prozessor durch ein spezielles Debug Interface gesteuert wird. Die Debug Support Unit (DSU) kontrolliert den Prozessor über diese Schnittstelle. Dies erleichtert das Debuggen auf der Zielhardware erheblich.

Die DSU wird als Slave an den AHB Bus angeschlossen und kann von allen AHB Mastern aus zugegriffen werden. Dies erlaubt ein Debuggen über eine Vielzahl von externen Schnittstellen: RS-232, JTAG, PCI, USB oder Ethernet. Wie auch der LEON3 ist die DSU3 Multi-Prozessor fähig und kann mit bis zu 16 LEON Prozessoren arbeiten. Folgendes Blockschema zeigt dies auf.

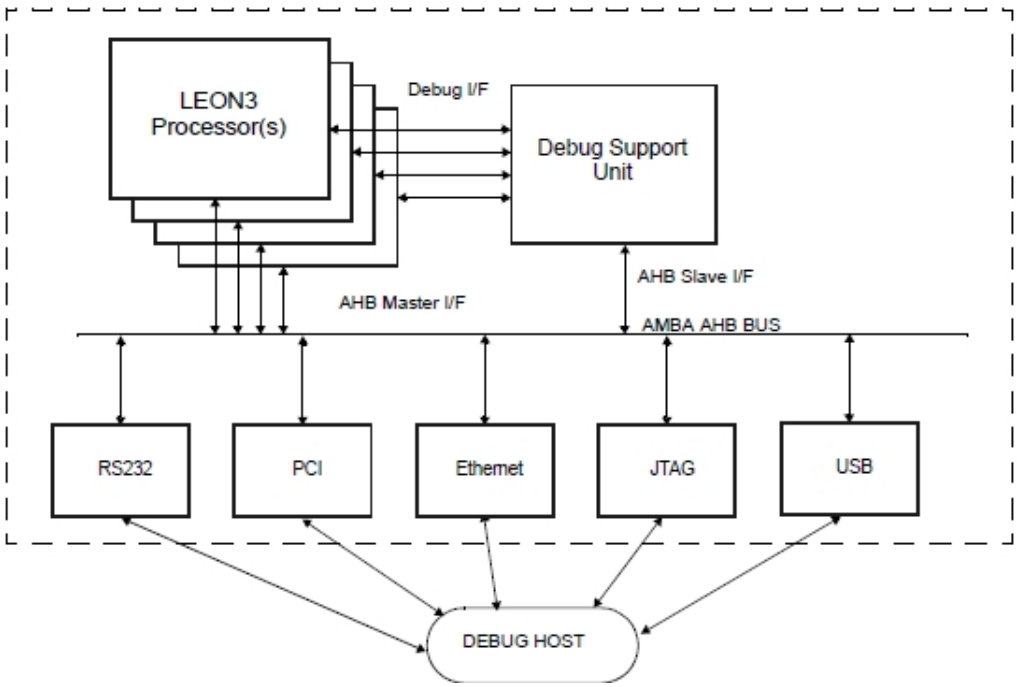


Abbildung 10: DSU3 Blockschema

Die AHB Master können so durch das DSU AHB Slave Interface auf die Prozessorregister und den Befehlstracebuffer zugreifen. Dies allerdings nur wenn der Prozessor sich im Debug Mode befindet. Der Debug Mode wird z.B. durch folgende Ereignisse aufgerufen:

- Erreichen eines Hardware oder Software Breakpoints
- Externes Signal DSUBRE
- Break Now (BN) Bit des DSU Kontrollregisters

Vorab muss der Debug Modus allerdings durch das externe Signal DSUEN aktiviert sein.

Beim Eintreten in den Debug Modus werden folgende Aufgaben durchgeführt:

- Program Counter werden in temporäre Register kopiert
- DSUACT Signal wird aktiviert
- Timer Einheit kann auf Wunsch gestoppt werden

Der Befehl, welcher den Debug Modus ausgelöst hat, wird nicht ausgeführt. Wird das BN Bit oder das DSUEN Signal deaktiviert, wird die normale Ausführung fortgesetzt.

Es besteht die Möglichkeit mit zwei getrennten Trace Buffern alle kürzlich durchgeführten AHB Transfers sowie Befehle zu kontrollieren. Beide Buffer sind Kreisbuffer und 128 Bit breit.

Die DSU besitzt eine Vielzahl an Konfigurations- und Statusregistern. Diese sind im GRLIB Auszug zusammen mit den Generics und Signalen in Beilage 6 zu sehen.

4.1.6 MCTRL IP Core

Der Speicherkontroller von Gaisler bietet die Möglichkeit an PROM, memory mapped I/O, synchronen Statischen RAM (SRAM) und synchronen dynamischen RAM (SDRAM) anzusteuern. MCTRL wird als AHB Slave an den AMBA Bus angeschlossen und besitzt drei Konfigurationsregister am APB Bus. Normal arbeitet der Controller mit 32 Datenbits, es ist aber auch möglich einen 8 oder 16 Bit Modus einzuschalten, beispielsweise für kleine Embedded Systems.

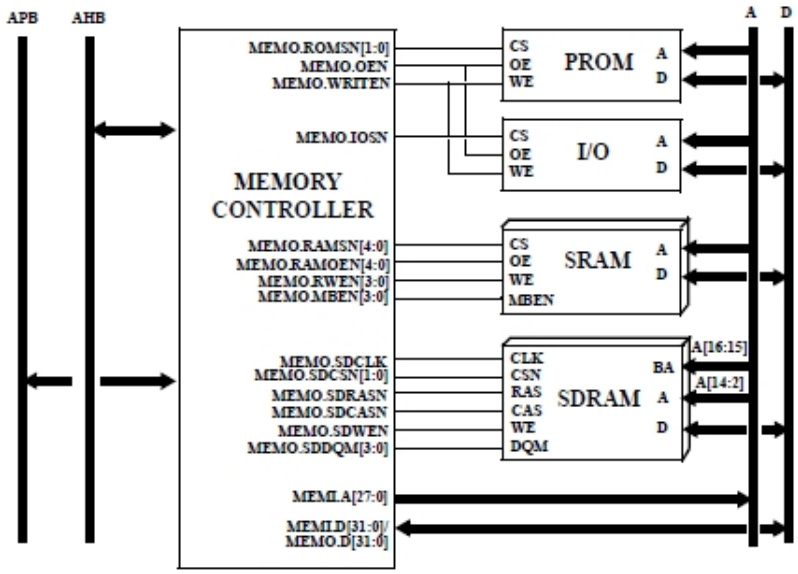


Abbildung 11: MCTRL Blockschema

Das Chip Select Decoding bietet Raum für zwei PROM Bänke, eine I/O Bank, fünf SRAM Bänke sowie zwei SDRAM Bänke. Für PROM, I/O und RAM werden drei getrennte Adressbereiche verwaltet. Neben normalen Lese- und Schreibzugriffen sind auch Burst Zugriffe möglich. Dies verbessert die Bandbreite bei Zugriffen auf sich nachfolgende Adressen.

Der SDRAM Controller wird durch den integrierten SDCTRL Core realisiert. Dieser erlaubt es zwei PC100/PC133 kompatible Bänke anzusteuern. Er erlaubt Grössen von 64M, 256M und 512M mit 8-12 Kolonnenadressbits und 13 Reihenadressbits. Der verbaute Micron SDRAM Speicher ist allerdings 128M gross, es geht somit die Hälfte des RAMs verloren. SDRAM Timings können über die Konfigurationsregister definiert werden.

In Beilage 7 sind alle Register, Generics und Signale aufgezeigt. Des Weiteren ist dort der IP Core im Detail beschrieben.

4.1.7 GRETH IP Core

Dies ist der Ethernet Controller Core von Gaisler. Er bietet Full und Half Duplex 10/100MBit/s Verbindungen an und kann an einen externen Physical (PHY) Layer Hardware Controller angeschlossen werden. Dazu bietet er Media Independent Interface (MII) und Reduced Media Independent Interface (RMII) Interfaces an sowie ein MMI Management Interface. Zudem wäre es auch möglich über Ethernet den Debug Modus per Ethernet Debug Communication Link (EDCL) zu handhaben. Dieser IP Core hat sowohl ein AHB Master Interface als auch ein APB Slave Interface. Das schnelle AHB Interface wird für den Datenverkehr eingesetzt. Intern wird dieses in einen Transmitter und einen Receiver Direct Memory Access (DMA) Kanal geteilt. Über die APB Schnittstelle sind die Konfigurationsregister erreichbar.

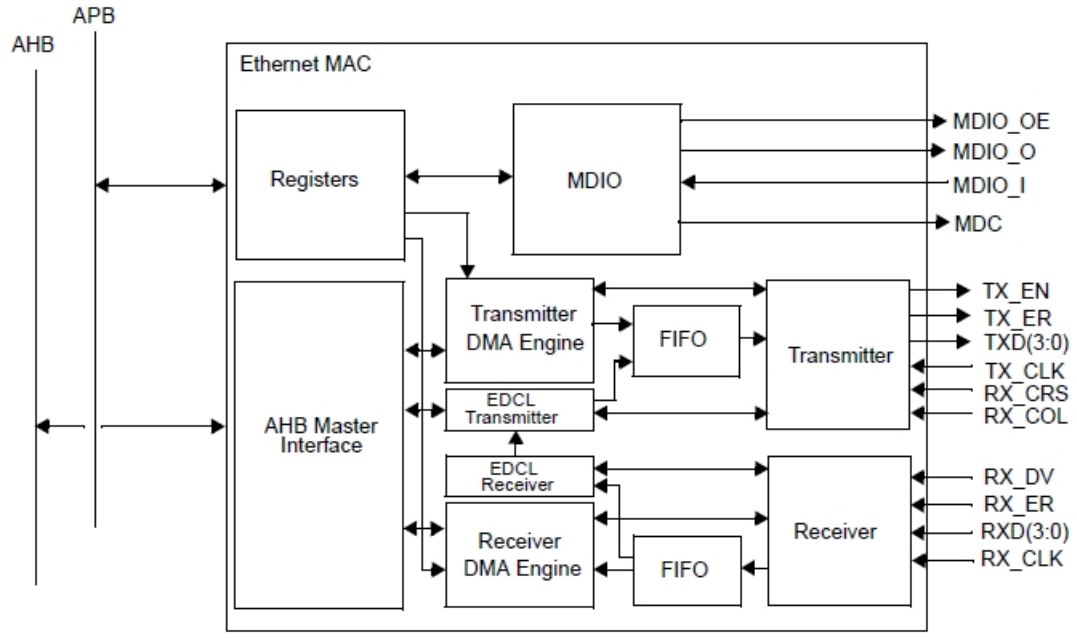


Abbildung 12: GRETH Blockschema

Wie in der obenstehenden Abbildung zu sehen ist, wird die Verbindung zum PHY Controller über eine Reihe von RX und TX Signale für die Daten sowie über ein Management Data Input/Output (MDIO) Interface für den Registerzugriff des PHY Chips realisiert. Die Hauptfunktion des Cores sind die beiden DMA Engines für den Empfang und den Versand der Daten. Die Konfiguration wird über APB Register vorgenommen. Per DMA hat der Core direkten Zugriff auf den Speicher für seine First In First Out (FIFO) Buffer. GRETH hat drei unterschiedliche Clock Domains: AHB Clock, Ethernet Receiver Clock sowie Ethernet Transmitter Clock. Die beiden Ethernet Clocks werden vom externen PHY Controller generiert. Alle drei Clocks sind unabhängig aber bereits intern im Core

synchronisiert.

Gaisler bietet auch für einige bekannte Real-Time sowie Linux Betriebssysteme Treiber für diesen Core an.

In Beilage 8 ist wiederum ein Auszug aus der GRLIB beigefügt. Darin sind alle Register, Generics und Signale zu sehen. Ebenfalls enthält dieser Auszug wichtige Informationen zum Empfang und Versand der Daten. Diese Informationen werden bei der Realisierung wichtig sein.

4.1.8 AHBUART Debug IP Core

Über ein serielles RS-232 Interface kann mit diesem Core auf die DSU3 zugegriffen werden. Daher ist dieser Core auch als AHB Master am AMBA Bus angeschlossen. Folgende Abbildung zeigt das Blockschema des AHBUART Cores:

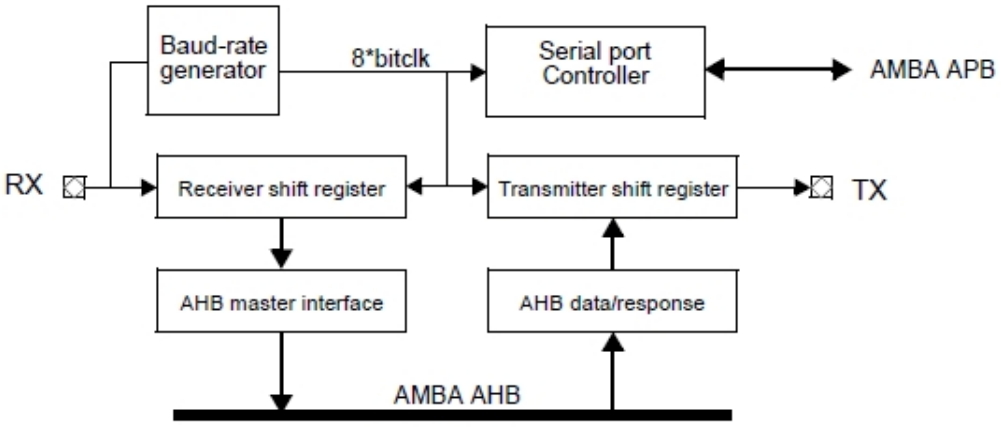


Abbildung 13: AHBUART Blockschema

Die Kommunikation findet über ein simples Protokoll statt. Befehle werden in ein Kontrollbyte, gefolgt von einer 32 Bit Speicheradresse und optional einem 16 Bit Datenwert bei einem Schreibvorgang eingeteilt. Die Daten werden stets in 8 Bit Paketen verschickt. Der Baud-Rate Generator kann per Software konfiguriert werden. Es ist aber auch möglich die Baud-Rate automatisch erkennen zu lassen. In Beilage 9 sind alle Register, Signale und Generics zu sehen.

4.1.9 APBUART IP Core

Dieser Core ist für die Serielle Kommunikation über RS-232 verantwortlich. Die Datenframes sind in ein Startbit, acht Datenbits, einem optionalen Paritätsbit sowie einem Stopbit aufgeteilt. Die Datenkommunikation zwischen APB Bus und UART kann über

interne FIFOs aber auch über Register vorgenommen werden. Ebenfalls kann eine Hardware Flusskontrolle (Handshake) aktiviert werden. Der Core bietet auch eine Interrupt Funktionalität an. Interrupts werden beispielsweise ausgelöst wenn der Eingangsdatenbuffer (FIFO oder Register) von leer auf voll wechselt oder wenn ein Paritätsfehler erkennt wird.

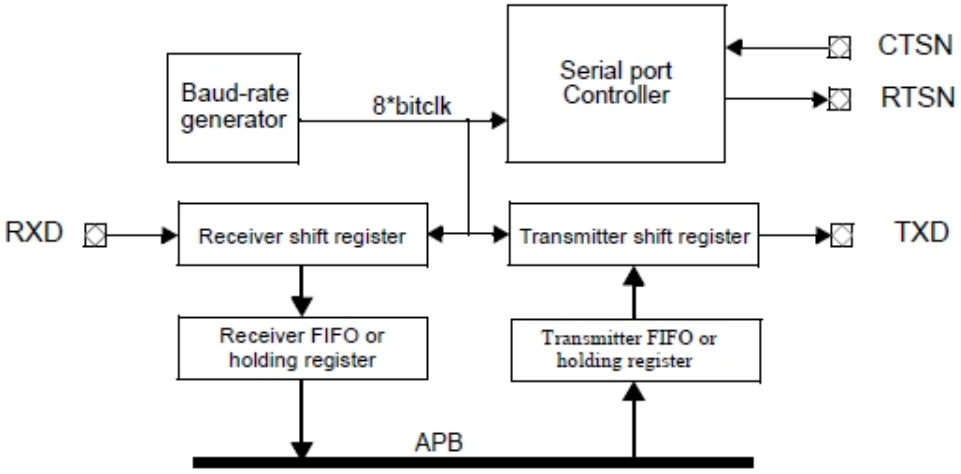


Abbildung 14: APBUART Blockschema

In Beilage 10 ist die Funktionalität des Cores detailliert beschrieben. Ebenfalls sind alle Konfigurationsmöglichkeiten diesem Anhang zu entnehmen.

4.1.10 IRQMP IP Core

IRQMP ist ein Multi-Prozessor fähiger Interrupt Controller. Gaisler hat in seiner AMBA Bus Implementation einen zusätzlichen Interrupt Bus hinzugefügt, welcher 15 Interrupts anbietet. Interrupts von allen AHB und APB Cores werden auf diesen Bus geroutet und an alle Cores übermittelt. IRQMP sitzt als APB Slave am AMBA Bus und überwacht den Interrupt Bus. Bei Empfang eines IRQs analysiert der Core den Interrupt, priorisiert ihn entsprechend und leitet ihn anschliessend an den Prozessor weiter. Bei mehreren IRQs wird der mit der höchsten Priorität zuerst verarbeitet. Der Core verfügt über zahlreiche Register für Interrupt Konfiguration und Status. Wird ein Interrupt auf dem Interrupt Bus (APBI.PIRQ) erkannt, wird das entsprechende Bit im Interrupt Pending Register gesetzt. Dieses kann erst durch ein Clear von der Software oder einem Interrupt Acknowledge vom Prozessor zurückgesetzt werden. Jedem Interrupt kann eines von zwei Levels (0 oder 1) zugeordnet werden. Dabei hat das Level 1 die höhere Priorität. Innerhalb jedes Levels wird jeder Interrupt anhand seiner Nummer priorisiert, d.h. dass Interrupt 15 die höchste Priorität hat.

In Beilage 11 ist dieser Core samt Registern, Generics und Signalen beschrieben.

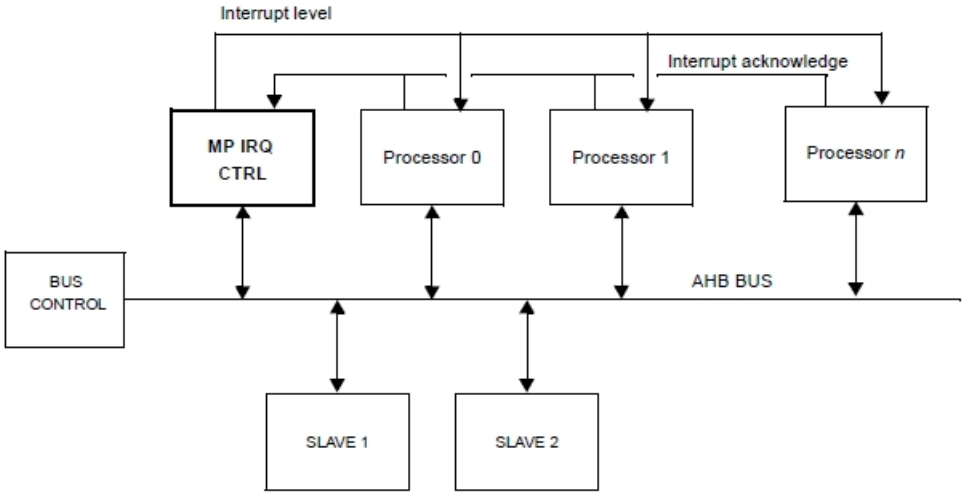


Abbildung 15: IRQMP Blockschema

4.1.11 GRGPIO IP Core

Dies ist der Core für General Purpose Interfaces. Die Portbreite kann zwischen 2 und 32 Bits definiert werden. Jedes Bit kann einzeln als Ein- oder Ausgang gewählt werden, ebenfalls ist es möglich für einzelne Linien einen Interrupt zu bestimmen. Für solche Konfigurationen stehen diverse Register zur Verfügung.

Folgende Abbildung zeigt den Aufbau einer I/O Leitung:

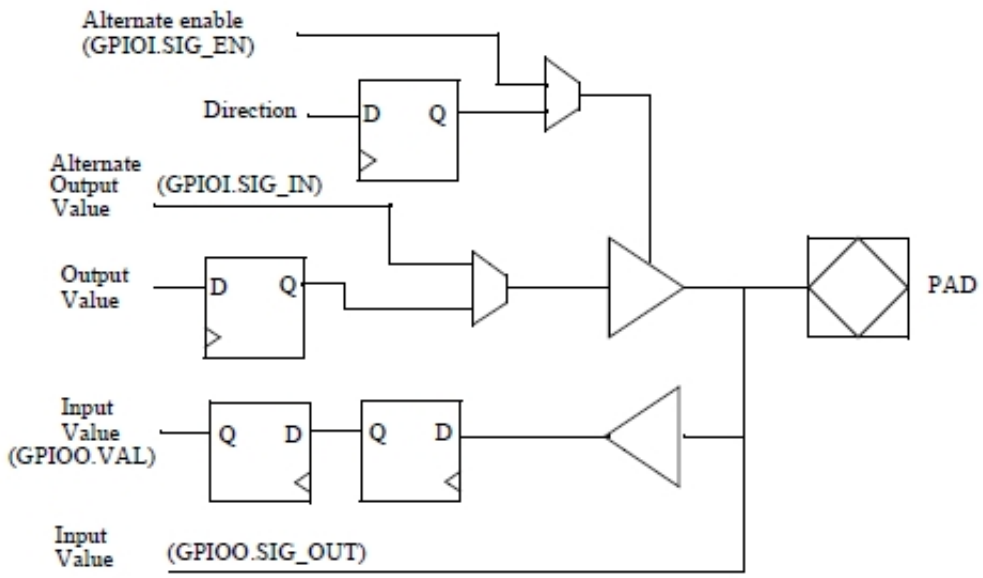


Abbildung 16: GRGPIO Blockschema

Jeder I/O Port ist als bi-direktionaler Buffer mit programmierbarem Output Enable implementiert. Die Eingangssignale sind alle mit zwei Flip-Flops synchronisiert um mögliche meta-stabile Zustände zu verhindern. Daten können über Register oder über Signale übermittelt werden.

In Beilage 12 ist ein Auszug aus der GRLIB zu diesem Core beigelegt.

4.2 USB Controller Core

Ein Universal Serial Bus (USB) Controller Core muss entworfen werden, da Gaisler keinen Open Source (GPL Lizenz) IP Core anbietet. Der verbaute Cypress EZ-USB FX2 Controller muss also über einen eigenen Controller angesteuert werden.

4.2.1 Cypress FX2

Der Cypress FX2 Controller ist ein Single Chip USB 2.0 Controller. Folgendes Block Diagramm zeigt den internen Aufbau des Chips:

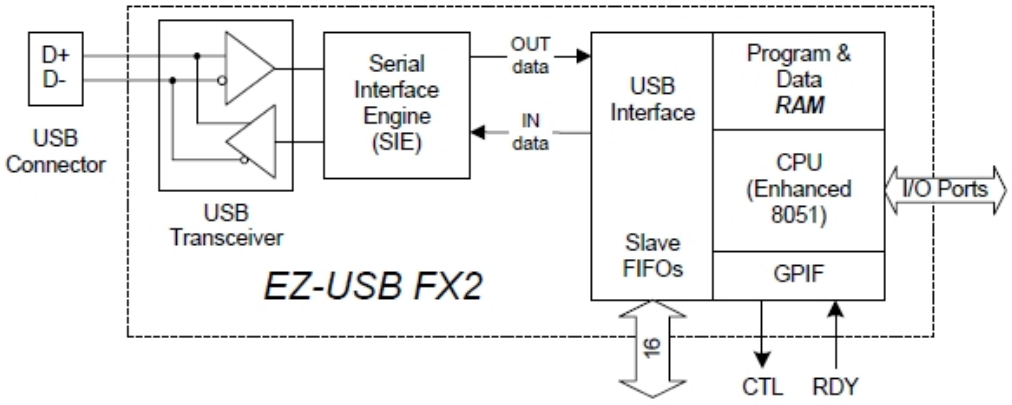


Abbildung 17: USB Core - FX2 Block Diagramm

Wie zu sehen ist, verfügt der FX2 über einen USB Transceiver (TX und RX), an welchem der USB Port angeschlossen ist. Dahinter befindet sich die Serial Interface Engine (SIE), diese wandelt die seriellen Daten auf den USB Leitungen in parallele Daten für das USB Interface bzw. die First In First Out (FIFO) Buffer um und umgekehrt. Ebenfalls ist eine erweiterte 8051 CPU inkl. lokalem Programm- und Daten- RAM verbaut. Die CPU wäre viel zu langsam um den High Speed USB 2.0 Verkehr zu managen. Deshalb wird sie primär nur für die Konfiguration des Chips eingesetzt und lässt anschliessend die FIFOs und eine externe Logik den Datenverkehr kontrollieren. Sie kann aber für langsamere USB Verbindungen das USB Protokoll handhaben. Ausserdem steht die CPU für allgemeine Aufgaben bereit und bietet I/O Ports, Counter, Timer usw. an.

Die FIFOs, welche für dieses Projekt das interessanteste Element darstellen, können über eine externe Logik oder über das interne General Purpose Interface (GPIF) gesteuert werden.

Die Aufgabe des Controllers ist es das FPGA_EBS_V2.0 Board, wenn es über USB an einen Computer (Host) angeschlossen wird, beim Host anzumelden und eine Datenkommunikation zu ermöglichen. Dazu lässt sich die Firmware des Chips über den verbauten 8051 Prozessor programmieren. Beispielsweise kann so die Geräte ID geändert werden, so würde beim Verbinden an einen PC ein individualisierter Gerätename angezeigt. Programmiert wird der 8051 Prozessor über ein mitgeliefertes Integrated Development Environment (IDE).

4.2.1.1 Endpoints Der FX2 besitzt sieben Endpoints. Gemäss der USB Spezifikation ist ein Endpoint eine Datenquelle. Genauer gesagt ein FIFO Buffer. Die Endpoints besitzen sehr unterschiedliche Buffer Grössen:

- 64 Bytes für Konfigurationstransfers
- 512 Bytes für langsamere Transfers
- 1024 Bytes für schnelle Transfers

Zudem können die schnellen Endpoints, je nach Bandbreitenbedürfnis, auch doppelt, dreifach oder vierfach gebuffert werden. So kann z.B. ein oder mehrere Buffer gefüllt werden während andere Buffer verarbeitet werden. Dies verbessert die Verfügbarkeit der Verbindung und somit auch die Bandbreite.

Die USB Spezifikation definiert IN und OUT Endpoints, folgende Grafik zeigt wie dies zu verstehen ist:

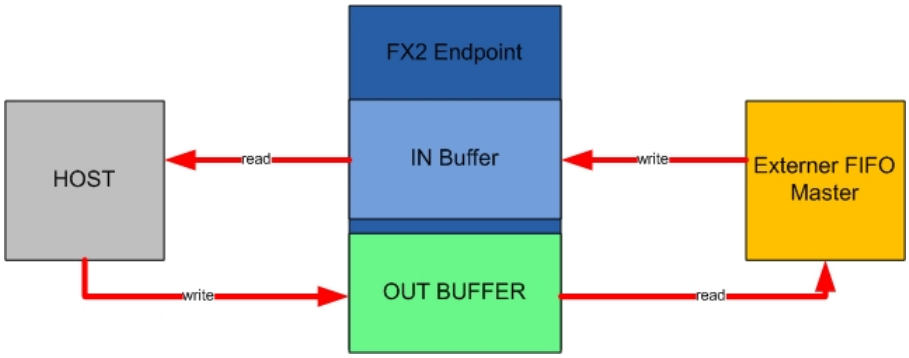


Abbildung 18: USB Core - Endpoints

Der Host schreibt in den OUT Buffer und liest vom IN Buffer. Die Namensgebung der Buffer bezieht sich also auf die Sichtweise des Hosts.

4.2.1.2 Slave FIFO Wie bereits erwähnt, können die FIFO Endpoints über eine externe Logik oder über ein FX2-internes GPIF kontrolliert werden. In diesem Fall bietet sich die externe Logik an, da diese in der FPGA sehr gut realisierbar ist und die Datenverarbeitung ebenfalls in der FPGA stattfindet.

Für externe Kontrolleinheiten bietet der FX2 ein Slave FIFO Interface an. Dies ist in untenstehender Abbildung zu sehen.

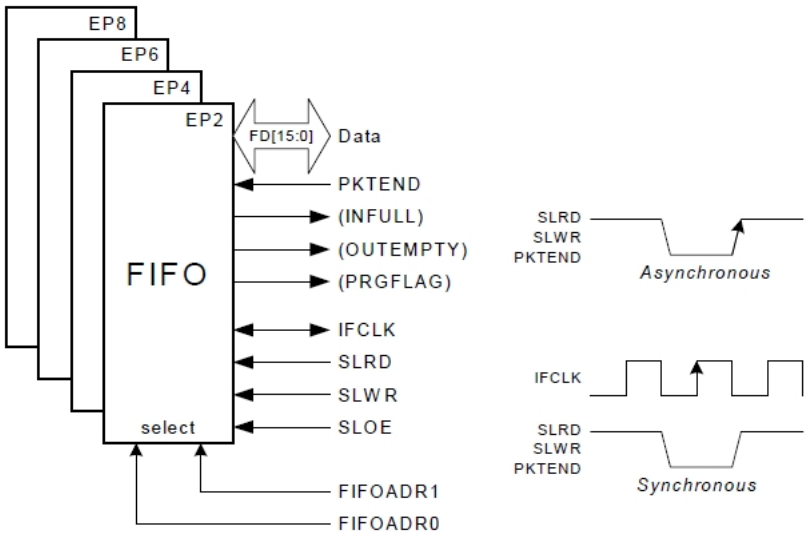


Abbildung 19: USB Core - Slave FIFO Interface

Das Slave Interface abstrahiert die FIFO Endpoints soweit, dass sie für eine externe Logik wie ganz normale FIFOs einsetzbar sind. So werden z.B. USB Paketgrössen für die externe Logik unwichtig. Der gewünschte FIFO lässt sich über zwei Adressbits wählen und kann anschliessend asynchron oder synchron mit einem Clock angesteuert werden. Dabei stehen Read und Write Strobes und auch ein Output Enable zur Verfügung. Die Daten werden über einen 16 Bit Bus übertragen. Das Interface bietet ebenfalls vier Status Flags an, diese können frei belegt werden. Wie im obenstehenden Schema zu sehen ist, werden diese z.B. für Full und Empty Flags verwendet. Die externe Logik kann nun auf diese Flags reagieren und die FX2 FIFOs wie normale FIFOs verwenden.

4.2.2 Aufbau Controller Core

Der zu entwickelnde USB Controller Core muss nun die Brücke zwischen LEON bzw. AMBA Architektur und dem FX2 Slave FIFO Interface schlagen. Es braucht also ein Interface zum AMBA Bus, eine Kontrolleinheit und alle Busse und Signale zum FX2 Interface. Folgende Abbildung stellt dies dar.

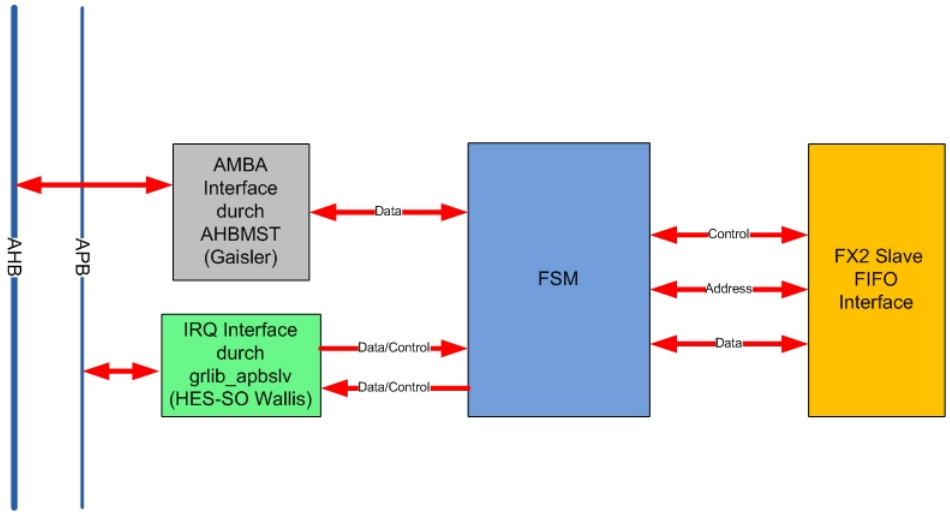


Abbildung 20: USB Core - Aufbau

Gaisler bietet für kundenentwickelte Cores einen AMBA Interface Controller an, dieser Core heisst *AHBMST* und abstrahiert den AMBA Bus sowie die von Gaisler hinzugefügten Funktionalitäten wie z.B. Plug & Play. Da USB 2.0 eine schnelle Übermittlung verlangt, wird der Core an den AHB Bus angeschlossen. Die Daten können von einer Zustandsmaschine (Finite State Machine, FSM) empfangen und weitergeleitet werden. Die FSM verfügt über alle Adress-, Kontroll- und Datensignale bzw. Busse des oben erwähten FX2 Slave Interfaces. So kann sie mit den Full und Empty Flags die FIFOs abfragen und anschliessend mit den Read oder Write Signalen ansteuern. Den gewünschten Endpoint FIFO adressieren und die Daten senden oder empfangen.

Damit die Übertragung stets synchron mit dem LEON Prozessor und der darauf laufenden Software bleibt, muss das restliche System über den Zustand des USB Controller Cores aber auch der FX2 FIFOs (full, empty) informiert werden. Dies wird über Interrupts realisiert. Die HES-SO Wallis bietet dafür ein IRQ Interface *grlib_ apbslv* an. Dieses abstrahiert nicht nur die Interrupts sondern den gesamten APB Bus. Es bietet Generics ähnlich wie die Gaisler Cores an, so dass sich auch ein APB Adressbereich einrichten lässt, z.B. für Steuer- und Statusregister. Die Zustandsmaschine und der LEON können über diese Register und durch Signalisation mit Interrupts kommunizieren.

4.3 FLASH Programmation

Mangels FLASH Support des MCTRL IP Cores muss für die Programmation des FLASH Speichers, d.h. für die permanente Speicherung der Software Applikation, eine zusätzliche FPGA Schaltung entwickelt werden. Diese Schaltung wird vorgängig in die FPGA geladen um den FLASH zu schreiben. Anschliessend kann die FPGA mit dem LEON System bestückt werden und die Software liegt schon bereit.

4.3.1 Aufbau

Untenstehende Abbildung gibt einen Überblick über das gewählte Vorgehen. In die FPGA wird über das JTAG Interface eine temporäre Schaltung geladen. Diese kann über den RS-232 Port die Daten empfangen und in den FLASH Speicher schreiben. Auf dem Host Computer wird die Memorydatei im SREC (.srec) Format [3] mit Hilfe eines Perl Scripts eingelesen und über ein eigens für diesen Zweck entworfenes Protokoll via RS-232 verschickt.

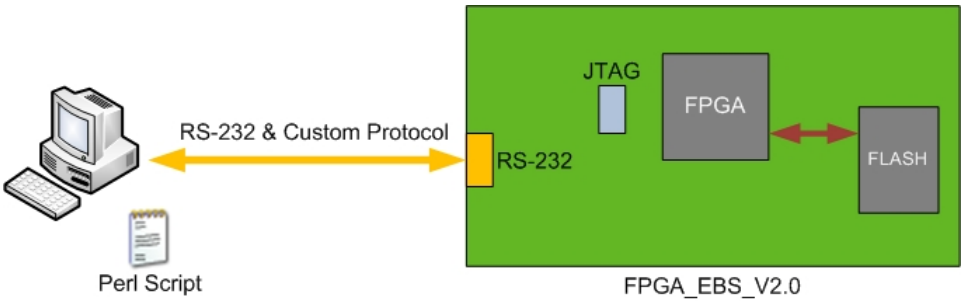


Abbildung 21: FLASH Programmation - Aufbau

[3] http://en.wikipedia.org/wiki/SREC_(file_format)

4.3.2 FPGA Schaltung

Wie in folgender Abbildung zu sehen ist, verfügt die Schaltung über ein RS-232 Interface für den Empfang und Versand von Daten. Die empfangenen Daten werden in dem Management Block analysiert. Dieser generiert wiederum Managementsignale hin zur Zustandsmaschine (Finite State Machine, FSM). Daten werden, falls diese im korrekten HEX Format vorliegen, weiter an die FSM übermittelt. Die FSM generiert anhand der erhaltenen Managementsignalen die Ansteuerungssignale für den FLASH, adressiert den Chip entsprechend und leitet die Daten weiter. Sie kann ebenfalls erhaltene Status (STS) Informationen an den Host zurück senden. Beim Lesen werden die Daten von der FSM aus dem FLASH gelesen und an die RS-232 Schnittstelle übermittelt.

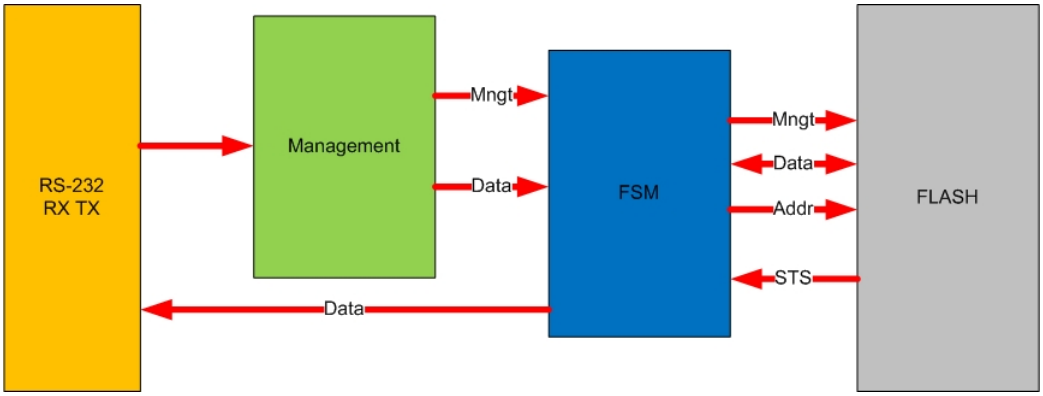


Abbildung 22: FLASH Programmation - FPGA

Bemerkung: Der FLASH befindet sich nicht in der FPGA selber. Der Block wurde zur besseren Veranschaulichung in das obere Schema aufgenommen.

4.3.3 Protokoll Definition

Das eingesetzte Protokoll wurde für diese Applikation entwickelt. Es ist die Grundlage für die Kommunikation zwischen dem Perl Script auf dem Host Computer und der Zustandsmaschine auf der FPGA.

Der Ablauf eines Transfers sieht wie folgt aus:

1. ERASE_COMMAND, fixer Wert, löst einen Löschvorgang des FLASH Speichers aus
2. START_ADDR_COMMAND, im Anschluss kann die Start Adresse, an welcher die Daten im Speicher geschrieben werden sollen, gesetzt werden
3. LENGTH_COMMAND, im Anschluss kann die Länge der zu übermittelnden Daten mitgeteilt werden
4. WRITE_COMMAND, im Anschluss können die Daten als HEX-Werte übermittelt werden
5. START_ADDR_COMMAND, im Anschluss kann die Start Adresse, an welcher die Daten im Speicher gelesen werden sollen, gesetzt werden
6. LENGTH_COMMAND, im Anschluss kann die Länge der zu lesenden Daten mitgeteilt werden
7. READ_COMMAND, liest den Speicher

Damit die Befehle und die Daten stets getrennt interpretiert werden können, werden in diesem Protokoll alle Werte als ASCII-Werte vom Host System übermittelt. Innerhalb der ASCII-Zeichen werden alle HEX-Werte (0-9, a-f, A-F) als Datenwerte reserviert. Alle anderen ASCII-Zeichen stehen für Managementbefehle zur Verfügung. Ebenfalls ist das Protokoll gross- und kleinschreibungsunabhängig.

Folgende Tabelle gibt einen Überblick über die Befehle und deren ASCII-Zeichen.

Befehl	**ASCII-Zeichen**
ERASE_COMMAND	x, X
START_ADDR_COMMAND	s, S
LENGTH_COMMAND	l, L
WRITE_COMMAND	w, W
READ_COMMAND	r, R
ERASE_ACK	o, O
ERASE_NACK	n, N
WRITE_ACK	v, V
WRITE_NACK	m, M
READ_ACK	p, P
READ_NACK	i, I

Tabelle 2: FLASH Programmation - Befehle

4.3.4 Einsatzmöglichkeiten des FLASH Speichers

Der FLASH Speicher dient in erster Linie dafür, Daten konstant speichern zu können. Auch wenn die Stromversorgung entfernt wird. Jedoch gibt es mehrere Möglichkeiten wie der Speicher eingesetzt werden kann bzw. was er für Daten speichern soll. Folgende Grafik zeigt drei mögliche Varianten auf.

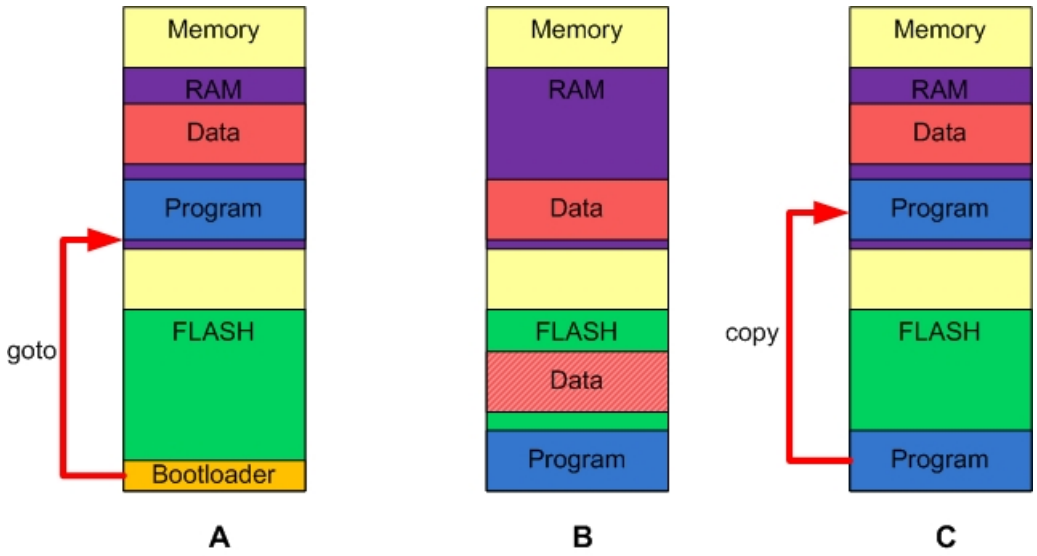


Abbildung 23: FLASH Programmation - Einsatzmöglichkeiten

- Variante A: Im FLASH Speicher befindet sich nur ein Bootloader. Das LEON System startet bei Adresse 0x0 und sucht dort nach Informationen wie es weiter geht. Der Bootloader enthält z.B. einen kleinen Code, der die CPU veranlasst im RAM nach den Programminformationen zu suchen. Dies kann während der Entwicklung praktisch sein, für ein Endprodukt allerdings nicht, da das Programm im RAM stets verloren geht sobald man das Board ausschaltet.
- Variante B: Im FLASH Speicher befindet sich direkt das Programm. Beim Start kann die CPU direkt das Programm ausführen. Die Daten können im FLASH oder im RAM gelagert werden, normal ist der RAM jedoch zu bevorzugen, da die Schreibgeschwindigkeiten dort viel schneller sind.
- Variante C: Im FLASH Speicher befindet sich wiederum das Programm. Dieses enthält zu Beginn Code, welcher einen Kopiervorgang in den RAM auslöst. Alternativ kann dies auch mit einem Bootloader durchgeführt werden. Im RAM werden während der Laufzeit also Programm und Daten vorhanden sein. Dies ist bei ungenügenden Lesegeschwindigkeiten des FLASH Speichers praktisch.

Während der Kompilierung und des Linkens kann mit Konfigurationsparametern bestimmt werden, welche Variante verwendet werden soll.

In diesem Projekt wird Variante B bevorzugt, da erstens die Speichergrössen von FLASH und RAM limitiert sind und ein getrennter Programm- und Datenspeicher daher praktisch ist und zweitens die Lesegeschwindigkeit des verbauten FLASH Speichers gut genug ist.

Während der Entwicklungsphase wird allerdings Variante A verwendet, da dies ein flexibleres Debuggen ermöglicht.

4.4 Debug Monitor & Demo Applikation

Aus zeitlichen Gründen wurde die Entwicklung des Debug Monitors und der Demo Applikation nicht während der Diplomarbeit durchgeführt. Für alle Debug Aufgaben wird weiterhin die gratis Testversion von GRMON verwendet.

Die Entwicklung der Demo Applikation wird jedoch nachgeholt um zur Präsentation der Diplomarbeit ein vorführbares System zeigen zu können.

5 Realisierung

In diesem Kapitel wird aufgezeigt wie das Projekt entwickelt wurde. Anhand dieses Abschnittes kann das Projekt nachvollzogen und reproduziert werden.

5.1 LEON System

Die Entwicklung des LEON Microcontroller Systems besteht aus mehreren Etappen. Die Aufgabe wurde somit in kleinere, übersichtlichere Blöcke unterteilt, welche wesentlich einfacher verständlich und realisierbar sind. Durch das Zusammenspiel dieser Blöcke entsteht ein System mit AMBA Bus, LEON Prozessor und diversen Peripherie Controllern.

5.1.1 Memory Map

Die erste Aufgabe bestand darin eine Memory Map zu entwickeln. Diese muss genügend Adressen für alle eingesetzten Komponenten aufweisen. Idealerweise mit etwas Reserve für zukünftige Ausbaustufen des System. Da es sich beim LEON um einen 32-Bit Prozessor handelt, adressiert der Prozessor in 4er Adresssprüngen, d.h. 0x0, 0x4, 0x8, 0xC usw. Pro Sprung können also vier Bytes (32-Bits) adressiert werden.

Die Memory Map ist auf der folgenden Seite dargestellt.

Der LEON Prozessor startet, bei Normalkonfiguration, bei Adresse 0x00000000. Deshalb wurde hier der FLASH Speicher angesetzt. Dieser nicht flüchtige Speicher beherbergt die Software Applikation, welche auf dem System ausgeführt wird. Von dort aus kann die Software während der Laufzeit in den RAM kopiert werden. Der FLASH Adressbereich reicht bis 0x1FFFFFFF und bietet sehr viel Reserve.

Im Anschluss daran folgt von 0x20000000 bis 0x2FFFFFFF der Bereich für extern angeschlossene I/O Komponenten. Beispielsweise um Register von anderen Chips ansprechen zu können, welche über eine GPIO oder Mezzanine Interface angeschlossen sind.

Von 0x40000000 bis 0x7FFFFFFF befinden sich die Adressen für den verbauten 16MByte SDRAM. Wie der FLASH wird auch der SDRAM von MCTRL verwaltet und angesteuert.

Der 800er Adressbereich wurde dem APB Bus zugeteilt, welcher vom APBCTRL verwaltet wird. Viele der IP Cores, AHB und APB, verfügen über Konfigurationsregister. Diese sind auch in den GRLIB Auszügen des Spezifikationskapitels ersichtlich. Solche Register müssen ebenfalls adressierbar sein und in der GRLIB werden alle Register, ausser die der DSU3, über ein APB Slave Interface angesprochen. Daher wurde im APB Bereich für jeden IP Core ein kleiner Adressbereich reserviert. 256 Adressen pro Core. Dies reicht

problemlos aus, da die meisten Cores nur zwischen drei und elf 32-Bit Register besitzen. Interessant ist der cfg-Bereich ab 0x800FF000. Dort wird die Plug & Play Tabelle für den APB Bus erstellt.

Ab 0x80100000 folgt die Debug Support Unit. Deren Register werden nicht über den APB Bus angesprochen, deshalb besitzt sie eine eigene AHB Adresse.

Am Ende des Adressbereichs folgt der AHB Abschnitt für AHBCTRL. Hier ist wiederum der Unterbereich ab 0xFFFFF000 interessant. Darin wird die AHB Bus Plug & Play Tabelle gespeichert.

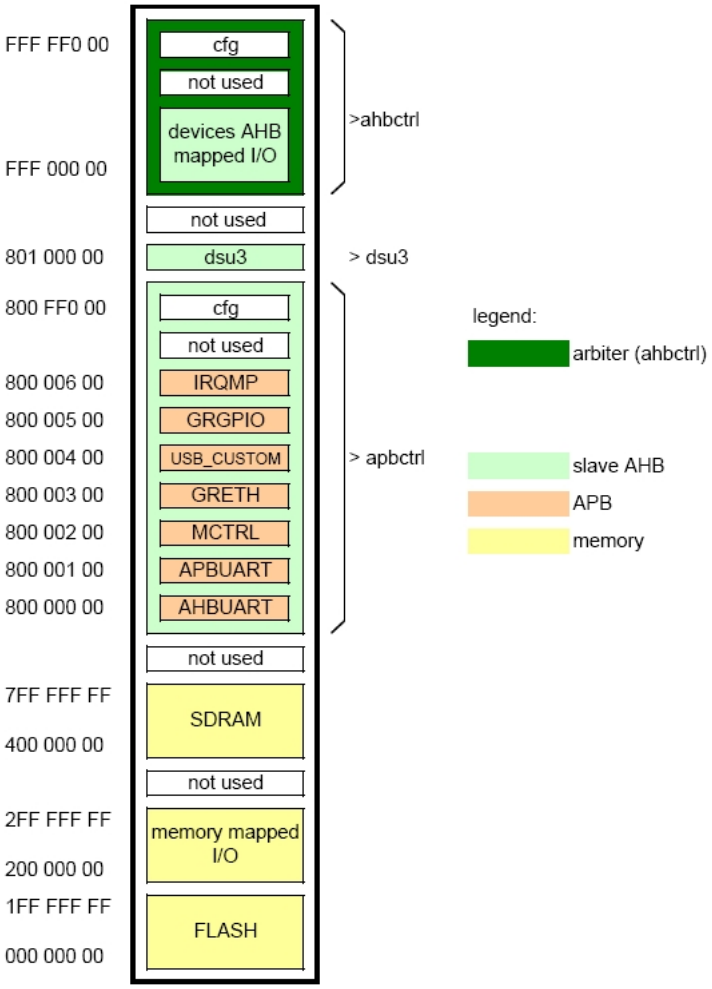


Abbildung 24: Memory Map

5.1.1.1 AMBA Adressierung Die Adressierung des AMBA Busses weist ein paar Besonderheiten auf. Diese werden nun erwähnt:

- die 12 Adress-MSBs (z.B. 0x800—–) definieren die AHB Adresse
- die 12 folgenden Bits (addr[19:8]) können für AHB I/Os, APB Slaves usw. verwendet werden
- die letzten 8 Bits sind für die genaue Adressierung, z.B. für ein Register, verfügbar

Falls die 8 LSBs für den geplanten Einsatzzweck nicht ausreichen, kann mit Hilfe von Masken der Bereich erweitert werden. Die Maske wird auf addr[19:8] angewandt und kann je nach Maskenwert den Bereich der Registeradressbits erhöhen. Die genaue Berechnung für solche Masken ist in Beilage 2 auf Seite 45 und 51 ersichtlich.

Für dieses System reichen die 8 LSBs für alle Komponenten aus, somit werden alle verfügbaren Masken auf 0xFFF gesetzt. Das heisst, alle 12 Bits (addr[19:8]) werden für die Adressierung des APB Slaves verwendet.

Ein IP Core Register, welches über ein APB Slave Interface erreicht werden kann, wird also in drei Etappen adressiert. Beispielsweise ein Register des GRETH unter 0x80000304:

- 0x**800**00304: die AHB Adresse entspricht dem APB Bereich (APBCTRL)
- 0x800**003**04: die APB Slave Bits identifizieren den GRETH
- 0x80000**304**: die LSBs adressieren schlussendlich das zweite Register dieses IP Cores

Alle Adressen und Masken werden über VHDL Generics definiert. Mehr dazu in Kürze.

5.1.2 AMBAdraw Vorbereitung

Als nächster Schritt kann nun, ausgehend von der Memory Map, das LEON System zusammengestellt werden. Dabei bietet sich das AMBAdraw Tool [4] an. Mit Hilfe dieses Tools können alle gewünschten GRLIB Komponenten grafisch an den AMBA Bus angeschlossen und konfiguriert werden.

5.1.2.1 Installation Die neuste Version von AMBAdraw kann unter dem in der Fussnote erwähnten Link gratis bezogen werden. Für diese Arbeit wurde Version 3.0.2 eingesetzt. Voraussetzung ist eine Perl Umgebung, da AMBAdraw für den Export des Designs auf Perl-Scripts setzt. Active Perl [5], in diesem Fall in Version 5.10, bietet dies auf einfache Art und Weise an.

AMBAdraw wird mit einem benutzerfreundlichen Installer installiert und bringt ein umfangreiches Handbuch inkl. Tutorial mit sich. Das Tutorial empfiehlt sich sehr um einen Einstieg in die Software zu finden.

5.1.2.2 GRLIB Vorbereitung und Import Gaisler bietet auf ihrer Website die neuste Version der GRLIB als Download an. Für diese Arbeit wurde Version 1.0.20 verwendet. Das .tar.gz Archiv, welches u.a. alle IP Core VHDL Source Codes enthält, kann an einen beliebigen Ort entpackt werden.

AMBAdraw verwendet für den HDL Designer Export ein vordefiniertes HDL Projekt. Dieses enthält zu Beginn ein System mit AMBA Bus, an welchem alle GRLIB IP Cores angeschlossen sind. Beim Export werden alle nicht im Design enthaltenen Cores entfernt. Dieses HDL Projekt muss nun mit der neusten, soeben geladenen GRLIB aktualisiert werden. Dazu muss unter *...\%AMBAdraw_Install_Dir%\source\template\HDS* das *AMBArchitect.hdp* Projekt geöffnet werden und anschliessend manuell alle Libraries aktualisiert werden. Folgende Libraries sind im Projekt bzw. in der GRLIB enthalten:

[4] http://ambadraw.ch.vu/
[5] http://www.activestate.com/activeperl/

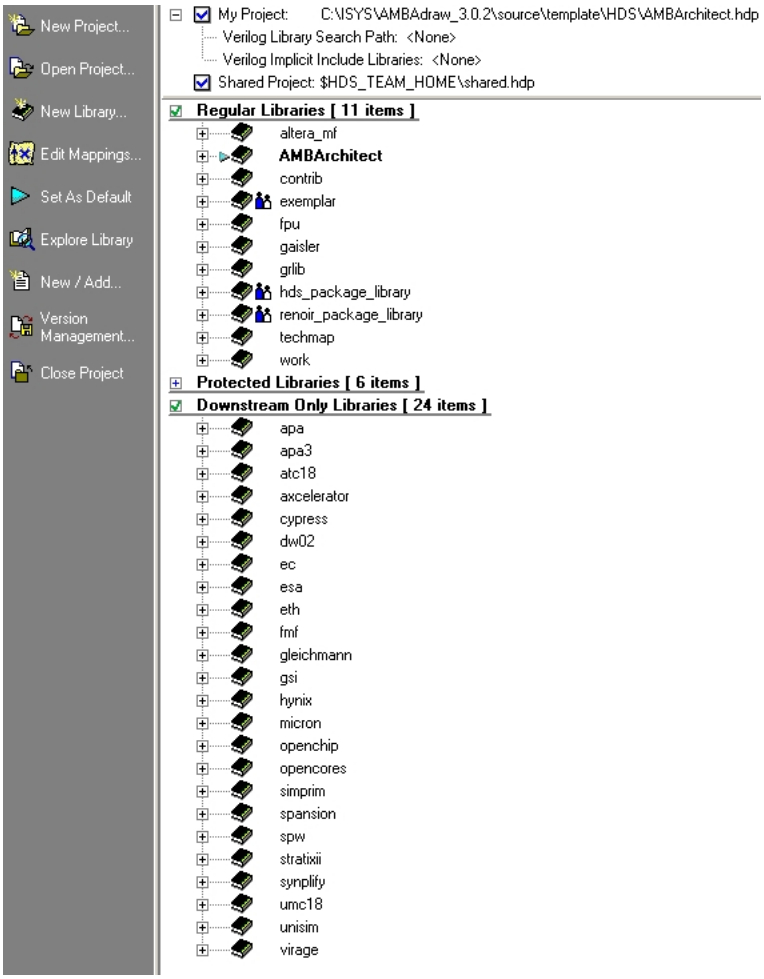


Abbildung 25: AMBArchitect.hdp Libraries

Die Libraries sind in normale (Regular) und Downstream Only eingeteilt. Downstream Only Libraries werden 1x generiert und sind anschliessend direkt verfügbar. Bei einer Generation des Projekts werden Objekte solcher Libraries nicht jedes Mal neu erstellt. Dies spart Zeit und Ressourcen. Der Typ einer Library kann jederzeit mit einem *Rechtsklick* ⇒*Change Library Type To* geändert werden.

Bei einem Update ist dies auch nötig. Denn es müssen die Libraries nicht nur hinzugefügt sondern auch neu generiert werden. Folgende Schritte sind für ein Update einer Library nötig und müssen für alle GRLIB Libraries durchgeführt werden:

1. Ändern des Library-Typs zu Regular (falls es sich um eine Downstream Only Library handelt)
2. Library wählen und Links in der Befehlsliste *New/Add...* wählen (siehe Abbildung oben)
3. Im neuen Fenster *Add existing HDL File(s) to an existing HDL Library* wählen und als Target Library die zu aktualisierende Library anklicken
4. Im nächsten Fenster zum zuvor entpackten GRLIB Verzeichnis (*...\%GRLIB_Install_Dir%\GRLib\grlib\lib*) wechseln und die zu aktualisierende Library wählen und mit *OK* bestätigen (siehe untenstehende Abbildung)
5. Anschliessend die Library im Design Manager öffnen und alle Elemente neu generieren
6. Ändern des Library-Typs zu Downstream Only (falls es sich um eine Downstream Only Library handelt)

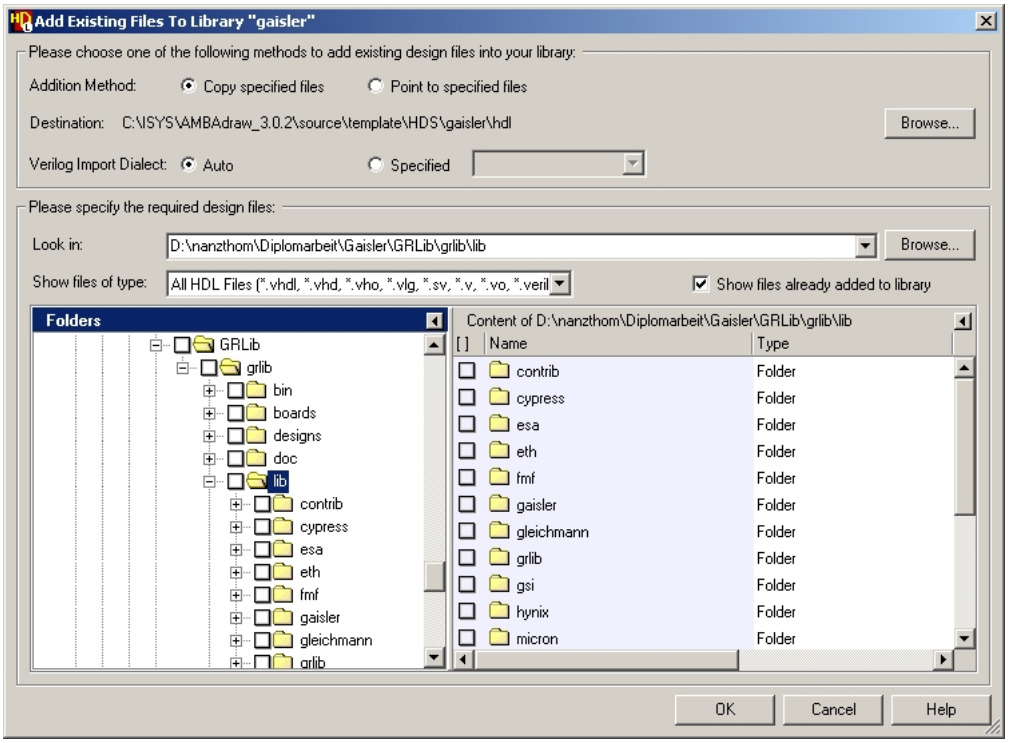


Abbildung 26: AMBArchitect.hdp Libraries Update

Der Template-Ordner lässt sich im Übrigen auch durch normales Kopieren und Einfügen sichern und wiederherstellen. Dies ist besonders im Falle eines Problems oder eines AMBAdraw Updates praktisch.

Letztendlich muss noch in AMBAdraw für jedes bereits erstellte Projekt die GRLIB aktualisiert werden. Dies wird unter *File* ⇒*Import Template Design...* und anschliessend mit *File* ⇒*Update with Template Design* gemacht. Dieser Vorgang ist im AMBAdraw Manual genauer erklärt.

5.1.3 AMBAdraw Entwurf und VHDL Generics

5.1.3.1 AMBAdraw Projekt erstellen Nach dem Start von AMBAdraw wird nun ein neues Projekt erstellt. Die Formateinstellungen können auf Wunsch angepasst werden. Wichtig sind jedoch folgende Parameter:

- Project Path: Zielverzeichnis des Projekt, hier wird auch der Export (z.B. als HDL Designer Projekt) abgelegt. Es sollte für jedes Projekt ein neues Verzeichnis angelegt werden, da bei einem Export alle Dateien überschrieben werden. Ein Pfad ohne Leerzeichen ist zu bevorzugen.
- Template Path: Verzeichnis des zuvor aktualisierten *AMBArchitect.hdp* Projekts.
- Tech: Zieltechnologie der verwendeten FPGA, in diesem Fall *Spartan3e*

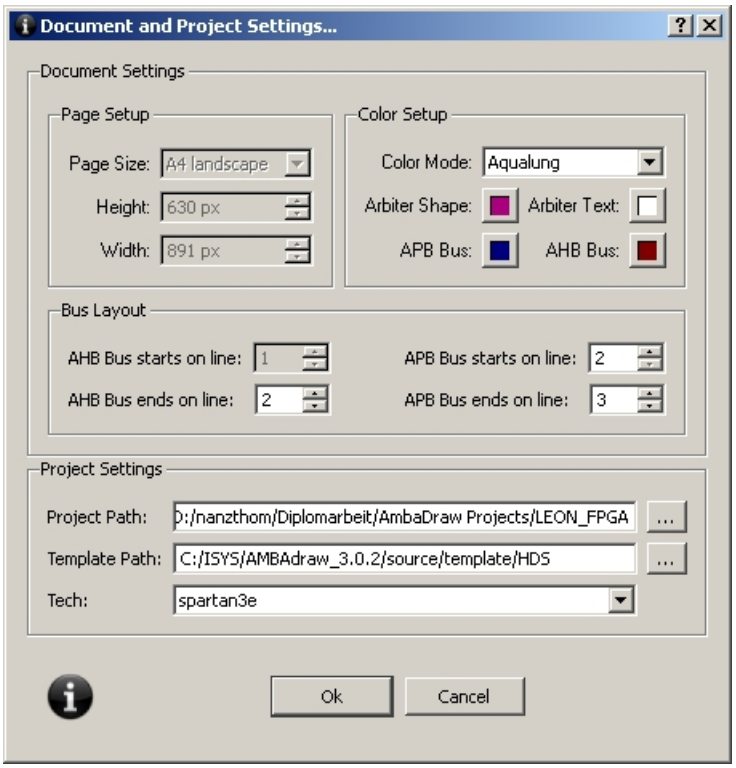


Abbildung 27: AMBAdraw Projekt Einstellungen

Achtung: Wie bereits erwähnt, überschreibt AMBAdraw bei einem erneuten Export alle vorhandenen Dateien innerhalb des *exportierten AMBArchitect.hdp* Projekts. Es besteht die Gefahr eines Datenverlustes! Daher sollte AMBAdraw nur für den ersten Rohentwurf verwendet werden. Spätere Änderungen, z.B. an den VHDL Generics, können in HDL Designer durchgeführt werden.

5.1.3.2 Design mit Hilfe von AMBAdraw Nun kann mit dem Entwurf des LEON Systems in AMBAdraw begonnen werden. An dieser Stelle soll nochmals das AMBAdraw Tutorial empfohlen werden. Beim Platzieren eines neuen Blocks erscheint folgendes Fenster:

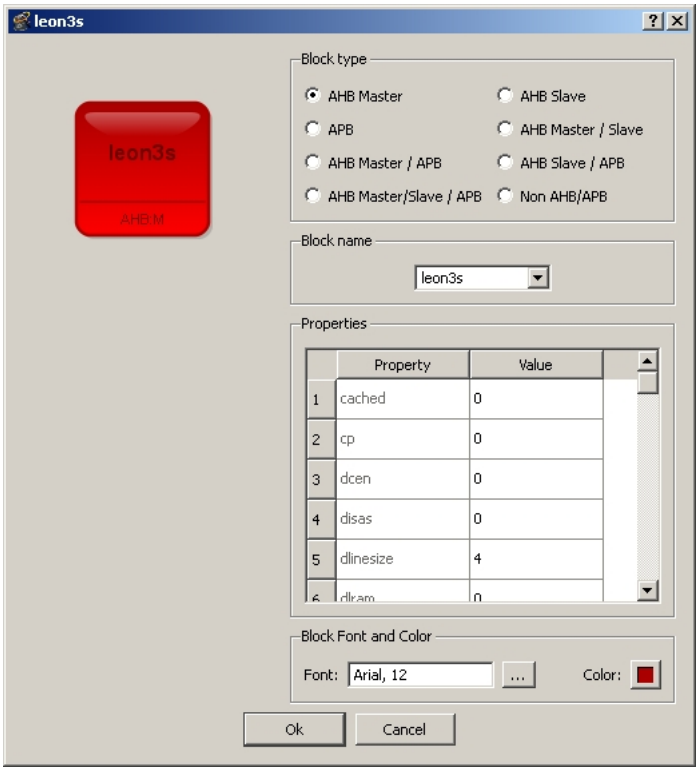


Abbildung 28: AMBAdraw Block hinzufügen

Als erstes muss der Block aus der GRLIB gewählt werden. In obenstehender Abbildung handelt es sich um den LEON3 IP Core. Unter Properties können alle IP Core relevanten VHDL Generics angepasst werden. Als Block Typ kann dessen Interface zum AMBA Bus gewählt werden. Wie welcher Block angeschlossen wird, ist in der GRLIB IP Core Dokumentation erwähnt (grip.pdf). Für alle im Rahmen dieser Diplomarbeit eingesetzten Cores sind die GRLIB Auszüge in Beilage 3 bis 12 angefügt. Der neue Block wird automatisch an den gewählten Bus angeschlossen.

Auf Wunsch können auch Kommentare grafisch hinzugefügt werden.

Das fertige Design sieht wie folgt aus:

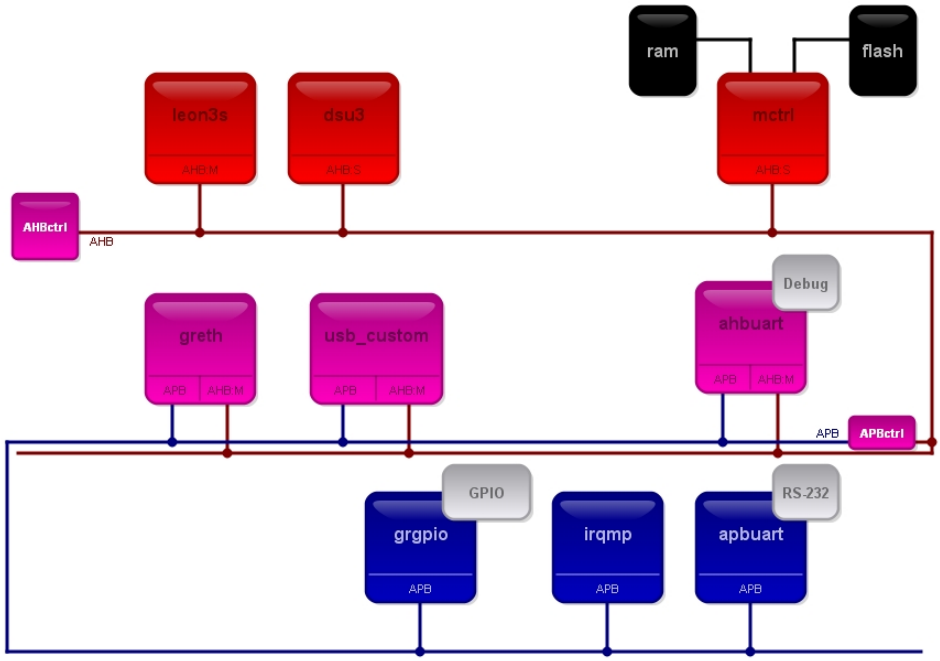


Abbildung 29: LEON System Entwurf

5.1.3.3 VHDL Generics Die GRLIB IP Cores sind so erstellt, dass die wichtigsten Funktionsparameter mit VHDL Generics bereits vor der Laufzeit konfiguriert werden können. Beispielsweise die Adressierung der Speicher, die Grössen der Caches aber auch Start- bzw. Resetwerte einiger Konfigurationsregister. Des Weiteren werden allgemeine AMBA Einstellungen, z.B. AHB/APB Index und Adressierung (siehe Memory Map), per VHDL Generics gesetzt.

Wie bereits erwähnt, lassen sich diese Generics auch mit AMBAdraw konfigurieren. Beim Hinzufügen eines Cores oder nach einem Doppelklick auf einen bereits hinzugefügten Core, können alle verfügbaren Generics eingestellt werden.

Nach dem Export legt AMBAdraw eine *config.vhd* Datei unter
...\%AMBAdraw_Project_Dir%\%Project_Name%\ambarchitect\hdl an. Darin sind nun für alle verwendeten Cores die aktuellen Generics gespeichert. Änderungen müssen nun in dieser Datei durchgeführt werden.

In Beilage 13 ist die *config.vhd* Datei zu sehen. Die Generics sind dort als VHDL Konstanten abgelegt. Die Namensgebung ist so gewählt, dass nach einem CFG Header der Name des Cores und zuletzt der Name des Generics auftritt. Während dem Export konfiguriert AMBAdraw die Generic Mappings der einzelnen Cores so, dass automatisch die korrekten Generics aus der *config.vhd* Datei gewählt werden. Ebenfalls wird die Datei auch automatisch in den einzelnen Schaltungen importiert.

Die Bedeutung der GRLIB Generics ist den Beilagen 3 bis 12 zu entnehmen. Wichtige Generics sind in Beilage 13 allerdings kommentiert.

An dieser Stelle werden nicht alle Generics erklärt. Jedoch nun folgend ein paar wichtige Informationen zur Konfiguration bestimmter Generics:

- HINDEX: AHB Master Index, muss eindeutig sein
- PINDEX: APB Slave Index, muss eindeutig sein
- HADDR: AHB Adresse, wichtig für DSU3 und APBCTRL, entspricht den 12 MSBs der AMBA Adressierung
- HMASK: Adressmaskierung (siehe Kapitel AMBA Adressierung), 0xFFF wählt alle 12 Bits aus
- PADDR: APB Adresse, entspricht den Bits addr[19:8] der AMBA Adressierung
- PMASK: Adressmaskierung (siehe Kapitel AMBA Adressierung), 0xFFF wählt alle 12 Bits aus

In der *config.vhd* Datei wurden auch zusätzliche Konstanten definiert, z.B. für die Busbreite (Anzahl Bits) der Adress- und Datenbusse. Dies bietet sich an, da die *config.vhd* als zentrale Konfigurationsdatei in der gesamten Schaltung verwendet werden kann. Es muss bei neuen Schaltungen oder Blöcken einzig die Datei importiert werden.

5.1.3.4 AHB und APB Indexierung Im vorhergehenden Kapitel wurde erwähnt, dass die HINDEX und PINDEX Generics eindeutig sein müssen. D.h., dass jeder Core seinen eigenen Wert haben muss. Dieser muss auf dem gesamten Bus einzigartig sein. AHB und APB Bus sind allerdings getrennt verwaltet.

Folgende Tabelle gibt eine Übersicht über die verwendete Indexierung:

IP Core	index			IRQ
	AHB		APB	
	master	slave		
LEON3	1	-	-	-
DSU3	-	2	-	2
MCTRL	-	3	2	-
GRETH	4	-	3	3
USB_CUSTOM	5	-	4	-
AHBUART	6	-	0	-
APBCTRL	7	-	-	-
APBUART	-	-	1	1
GRGPIO	-	-	5	-
IRQMP	-	-	6	X

X special case (See: 47.2 Operation, grlip.pdf - page 395)

Tabelle 3: AHB & APB Indexierung

5.1.3.5 HDL Designer Export Nun da alle gewünschten IP Cores platziert und deren Generics entsprechend gesetzt sind, kann ein Export gestartet werden. Dazu muss in AMBAdraw *File* ⇒*Export* ⇒*HDL Designer* gewählt werden. AMBAdraw startet daraufhin ein Export Script, welches im Projektverzeichnis das HDL Designer Projekt erstellt.

Nach dem Export lässt sich dieses durch einen Doppelklick auf *AMBArchitect.hdp* öffnen.

5.1.4 Input / Output Zuweisungen

AMBAdraw hat nun eine Haupt-Library innerhalb des HDL Projekts erstellt, diese heisst *ambarchitect*. Darin befindet sich die Hauptschaltung *toplevel*. Diese enthält alle gewählten Gaisler IP Cores, deren Generic-Zuweisungen und die AMBA Bus Signale.

5.1.4.1 Umbenennen Top-Level Die Schaltung *toplevel* in *ambarchitect* wird im LEON System nicht das oberste Niveau (Top-Level) sein. Um eine Synthese mit einer einzigen Datei (single file) sicherstellen zu können, muss das toplevel umbenannt werden. Der Grund hier für liegt darin, dass bei einer single file Synthese innerhalb des Files jedes Element seinen eindeutigen Namen besitzen muss.

Der neue Name der Schaltung lautet für dieses Projekt *toplevel_ arch*.

5.1.4.2 Gaisler IP Core I/Os Alle Gaisler IP Cores sind nun als HDL Designer Blöcke (struct) in der Schaltung zu sehen und bereits korrekt an den AMBA Bussen (AHB und APB) angeschlossen.

Stellvertretend für alle Cores wird nun GRETH näher betrachtet.

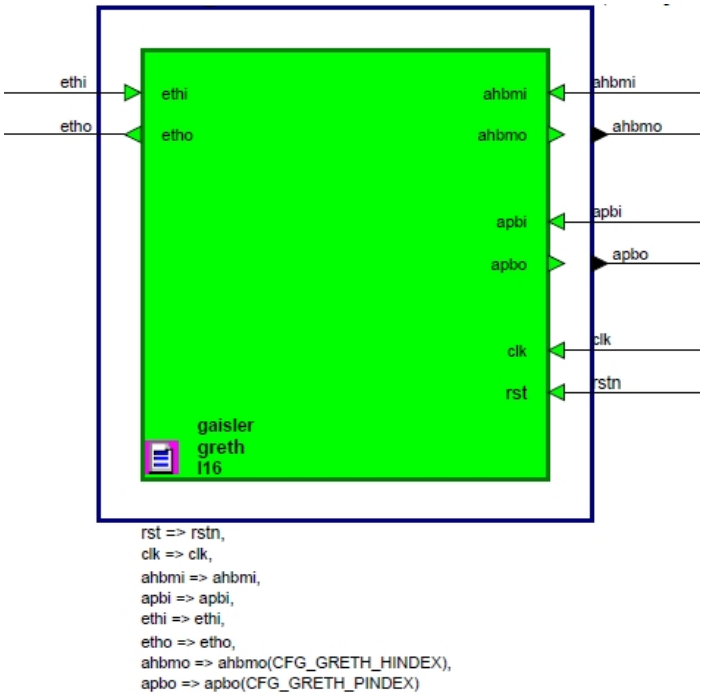


Abbildung 30: GRETH HDL Designer struct

In der obenstehenden Abbildung ist der GRETH Core dargestellt. In der unteren linken Ecke sieht man in der ersten Zeile die Library, aus welcher der Core stammt. In der zweiten Zeile dessen Namen und in der dritten Zeile den Namen innerhalb der Schaltung. Jeder Block muss seine eindeutige Bezeichnung innerhalb des aktuellen Niveaus/Schaltung haben, dies ist beispielsweise beim Wiederverwenden von Blöcken ausschlaggebend.

Die grünen Dreiecke sind die Inputs und Outputs (I/Os) des Blocks. Die Richtung des Dreiecks steht zugleich für die Richtung der Signale. Es sind auch bi-direktionale Signale möglich, darstellt durch ein gedrehtes Quadrat.

Der blaue Rahmen um den Core ist dessen PortMap Frame. Darin werden die Block I/Os mit den Signalen oder Bussen der Schaltung verbunden. Interessant ist hier das Mapping für AHB Master Output (ahbmo) und den APB Output (apbo). Mit Hilfe des HINDEX und PINDEX Generics wird nun das Signal auf dem Bus gewählt, welches nur für diesen Core reserviert ist.

5.1.4.3 Gaisler IP Core Signal Routes Wie in den Beilagen 3 bis 12 zu sehen ist, sind die I/O Signale der Gaisler Cores alle als VHDL Records implementiert. Das heisst unter einem Namen, z.B. etho, sind mehrere Signale und Busse zusammengefasst. Hardwaremässig ist dies aber bei einer FPGA nicht möglich, da jeder I/O seinen eigenen Pin der FPGA erhalten muss. Es muss also das VHDL Record in einzelne Signale bzw. Busse aufgeteilt werden.

Wiederum wird dies anhand von GRETH gezeigt. Es wird neben dem IP Core ein Embedded-Block (gelb) hinzugefügt. Dieser wird Signal Routes Block benannt und beinhaltet VHDL Code, welcher die Aufteilung der Records in Signale und Busse ermöglicht. Daran angeschlossen werden nun alle Signale/Busse inkl. Ports. Die Ports, blaue, gerichtete Pfeile, sind nun die endgültigen I/Os dieser Schaltung. Mit diesen kann später weitergearbeitet werden, mehr dazu später.

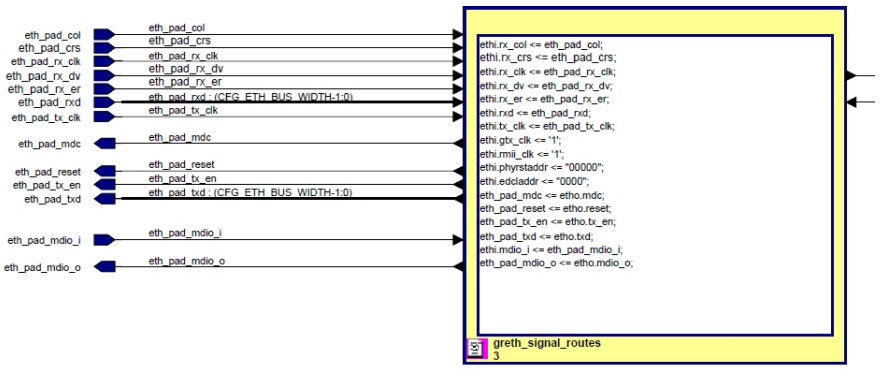


Abbildung 31: GRETH Signal Routes

Zu beachten sind die Namen der Signale. Alle werden zusätzlich mit einem *_pad* versehen. Mehr dazu später.

Der gesamte VHDL Code von *toplevel_arch*, inkl. allen Signal Routes ist in Beilage 14 zu sehen. Hier sind vor allem die Zuweisungen von MCTRL wichtig, dazu später mehr.

5.1.5 16 Bit Speicher Anpassungen

Sowohl der verbaute FLASH Chip als auch der SDRAM Chip verfügen über ein 16 Bit Dateninterface. Der LEON Prozessor und der AMBA Bus sind allerdings 32 Bit breit. Der Speichercontroller, MCTRL, ist allerdings in der Lage die 32 Bit Zugriffe seitens LEON/AMBA in zwei 16 Bit Zugriffe auf den Speicher zu übersetzen. Dazu sind folgende Schritte nötig:

- *config.vhd* Konstante CFG_MCTRL_RAM16 = 1
  Dies setzt das entsprechende MCTRL Generic und teilt dem Speichercontroller mit, dass er im 16 Bit Modus arbeiten soll.
- Signalzuweisung in MCTRL Signalroutes Block:
  memi.bwidth <= CFG_MCTRL_REG_ROMBWIDTH;
  Mit CFG_MCTRL_REG_ROMBWIDTH = "01" in *config.vhd*. Dieses Signal wird beim Systemstart automatisch in die Bits PROM WIDTH des Registers MCFG1 kopiert. So wird der PROM, in diesem Fall der FLASH, direkt im 16 Bit Modus gelesen.
- Prozess *memiproc* im MCTRL Signalroutes Block:
  Damit werden die Daten auf dem Eingangsbus für FLASH (16 MSBs) und SDRAM (16 LSBs) gemappt, siehe Beilage 14
- Signalzuweisung im MCTRL Signalroutes Block:
  mctrl_pad_data_out <= memo.data(15 downto 0);
  Die 16 wertniedrigsten Bits werden für die ausgehenden Daten verwendet.
- Keine Änderungen am Adressbus.

Nach diesen Anpassungen ist der 16 Bit Betrieb problemlos möglich. Dies gilt für die Simulation aber auch für den Betrieb auf dem Board. Die Simulation kann, wie in diesem Projekt, mit dem Modell des Intel FLASHs aber auch mit dem Gaisler SRAM16 Block durchgeführt werden. Kommt der SRAM16 Block zum Einsatz, sind folgende Punkte zu beachten:

- Index 4 und 5 der internen SRAM Blöcke, d.h. das index Generic von SRAM16 muss auf 2 gesetzt werden
- *config.vhd* Konstante CFG_MCTRL_SRBANKS = 2 um zwei Speicherbänke zu verwenden.

5.1.6 SDRAM & MCTRL Konfiguration

5.1.6.1 SDRAM Einführung Ein Synchronous Dynamic Random Access Memory, kurz SDRAM, lässt sich vereinfacht durch eine zweidimensionale Matrix darstellen. In jeder Zelle der Matrix lassen sich Daten speichern. Adressiert werden diese Zellen über Reihen- und Zeilenadressen. In der Praxis sind die Speicher viel komplexer aufgebaut, jedoch funktioniert die Adressierung weiterhin in dem Reihen und Zeilen System. Der eingesetzte Chip verfügt über 4096 Reihen und 512 Zeilen mit je 16 Bits. Zudem gibt es mehrere Bänke mit unabhängigen Adressdecodern. So kann pro Bank eine Zeile aktiv sein und es können mehrere Bänke zeitgleich verwendet werden. So kann z.B. der Speichercontroller von einer Bank lesen und für eine andere Bank bereits die Adressen vorbereiten. Der verwendete SDRAM besitzt vier solcher Bänke, der Gaisler Speichercontroller (MCTRL) kann allerdings nur zwei davon ansteuern. Das heisst, dass von den 16MByte nur 8MByte verwendet werden können.

Der Speicher besitzt auch einen eigenen Takt, typischerweise 100 oder 133MHz. In diesem Fall wird jedoch der Systemtakt von 66MHz auch für den Speicher verwendet. Es wäre allerdings möglich mit einem von Gaisler entwickelten Block (clkgen) ein synchronisiertes, höheres SDRAM Taktsignal erstellen zu lassen.

Ein SDRAM besitzt folgende Eingangssteuersignale:

- CKE: Clock Enable
- RAS: Row Address Strobe
- CAS: Column Address Strobe
- WE: Write Enable
- CS: Chip Select

Bei dem verwendeten Speicher von Micron sind all diese Signale zudem aktiv tief. Im Gegensatz zu normalem DRAM, haben bei Synchronen DRAM die oben aufgelisteten Signale eine andere Funktion. Bei DRAM wird z.B. die aktuelle Adresse bei aktiviertem RAS als Reihenadresse interpretiert. Bei SDRAM wird die Kombination aller Eingangssignale bei steigender Clock-Flanke ausgewertet und dann entschieden wie die Adresse zu interpretieren ist. Folgende Tabelle gibt einen Überblick über die Befehle und deren Signalzustand für den Micron SDRAM.

NAME (FUNCTION)	CS#	RAS#	CAS#	WE#	DQM	ADDR	DQs	NOTES
COMMAND INHIBIT (NOP)	H	X	X	X	X	X	X	
NO OPERATION (NOP)	L	H	H	H	X	X	X	
ACTIVE (Select bank and activate row)	L	L	H	H	X	Bank/Row	X	3
READ (Select bank and column, and start READ burst)	L	H	L	H	X	Bank/Col	X	4
WRITE (Select bank and column, and start WRITE burst)	L	H	L	L	X	Bank/Col	Valid	4
BURST TERMINATE	L	H	H	L	X	X	Active	
PRECHARGE (Deactivate row in bank or banks)	L	L	H	L	X	Code	X	5
AUTO REFRESH or SELF REFRESH (Enter self refresh mode)	L	L	L	H	X	X	X	6, 7
LOAD MODE REGISTER	L	L	L	L	X	Op-Code	X	2
Write Enable/Output Enable	–	–	–	–	L	–	Active	8
Write Inhibit/Output High-Z	–	–	–	–	H	–	High-Z	8

Tabelle 4: SDRAM Befehle

Die Reihenadresse und die Speicherbank werden also bei einem ACTIVE Befehle gesetzt. Die Adresse, die bei einem READ oder WRITE Befehl anliegt, ist die Zeilenadresse.

Nach jedem durchgeführten Vorgang, sei es ein Einzelzugriff oder ein Burstzugriff auf mehrere aneinander liegende Zellen, ist ein PRECHARGE Befehl nötig. Dieser deaktiviert die zuvor gewählte Bank und Reihe wieder, so dass der Speicher für einen neuen Zugriff bereit ist.

Die REFRESH Befehle spielen ebenfalls eine entscheidende Rolle. Diese steuern den Speicher in periodischen Intervallen kurz an, damit die Ladekapazitäten in den Kondensatoren aktualisiert werden können. Die sonst auftretenden Leckströme können die Ladung so verändern, dass der Wert nicht mehr korrekt interpretierbar ist.

MCTRL kontrolliert diese Befehlsfolge automatisch. Es müssen allerdings gewisse Timings eingehalten werden.

Timing	Beschreibung	Wert
CL /CAS	Zeit, welche zwischen der Absendung eines Lesekommandos und dem Erhalt der Daten vergeht.	2
tRCD	Zeit, die nach der Aktivierung einer Wortleitung (Activate) verstrichen sein muss, bevor ein Lesekommando (Read) gesendet werden darf.	2
tRP	Zeit, die nach einem Precharge-Kommando mindestens verstrichen sein muss, bevor ein erneutes Kommando zur Aktivierung einer Zeile in der gleichen Bank gesendet werden darf.	2

Tabelle 5: SDRAM Timings

Die Werte beziehen sich immer auf die Anzahl Taktperioden des SDRAM Taktes.

5.1.6.2 MCTRL Konfiguration Gewisse grundlegende Einstellungen lassen sich per Generic oder Eingangssignalen vornehmen. Diese konfigurieren jeweils ein paar Bits in den Konfigurationsregistern des Controllers, so dass ein Starten des Systems möglich ist. Jedoch müssen die restlichen Bits der Konfigurationsregister per Software genau definiert werden. Sonst ist z.B. ein Zugriff auf den SDRAM nicht möglich. MCTRL bietet drei Konfigurationsregister an, deren Konfiguration wird nun aufgezeigt.

MCFG1 Diesem Register wird der Wert *0x1000010F* zugeteilt. Primär wird dadurch der PROM Controller, in diesem Fall für den FLASH, für einen 16-Bit Bus eingestellt. Ebenfalls werden Schreibzugriffe auf den PROM Bereich deaktiviert.

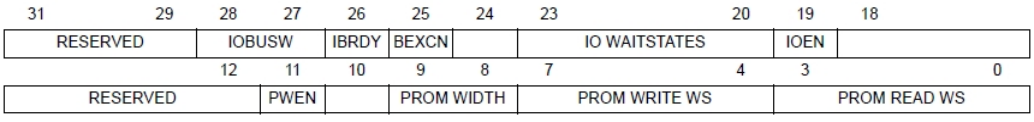


Abbildung 32: MCFG1 - Register für PROM Einstellungen

MCFG2 Für den SDRAM ist dies das zentrale Register. Hier können die oben erwähnten Timings eingestellt werden. Ebenfalls wird der SDRAM Controller für 4MByte Bänke und 512 Zeilen eingestellt. Auch wird der SDRAM aktiviert und zugleich SRAM deaktiviert,

so wird der SDRAM Bereich in die untere Hälfte des RAM Bereichs gemappt, d.h. ab 0x40000000. In dieses Register wird der Wert *0x9030605A* geschrieben.

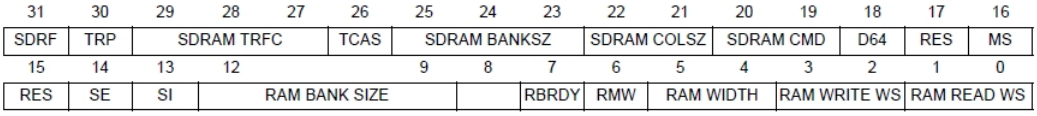


Abbildung 33: MCFG2 - Register für SRAM/SDRAM Einstellungen

MCFG3 Hier kann das REFRESH Intervall definiert werden. Gemäss Micron Datenblatt muss ein Refresh alle 15.625us erfolgen. Dies ergibt mit der in Beilage 7 ersichtlichen Formel einen exakten Refreshwert von 1030.25. Um etwas Spiel zu haben, wird ein Wert von 1024 gewählt. Somit wird dieses Register mit *0x00400000* eingestellt.

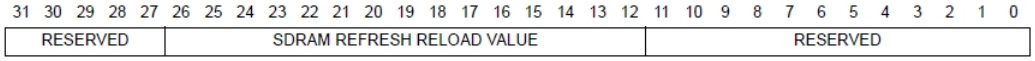


Abbildung 34: MCFG3 - Register für SDRAM Refresh Einstellungen

In Beilage 7 sind die Definitionen der einzelnen Parameter zu sehen.

Softwareimplementation Die Register können zu Beginn der Applikation wie folgt geschrieben werden. Dieses Vorgehen kann übrigens für alle anderen Register auch verwendet werden.

Algorithm 1 MCTRL Registerkonfiguration

```
#define MEMCTRLAREA (0x80000200)
int main(int argc, char *argv[])
{
   //define pointer to memory control area
   volatile unsigned int *memctrl = (volatile unsigned int *)MEMCTRLAREA;
   //MCTRL1
   memctrl[0] = 0x1000010F;
   //MCTRL2
   memctrl[1] = 0x9030605A;
   //MCTRL3
   memctrl[2] = 0x00400000;
   //program code
}
```

Die Register sind alle 32 Bit gross, somit kann mit einem Integer Pointer auf die einzelnen Register zugegriffen werden. Dabei ist zu beachten, dass die Indexierung bei 0 beginnt.

SDRAM Generics Folgende wichtige Generics müssen für den SDRAM Betrieb gesetzt werden. In Beilage 13 ist ein Überblick über alle MCTRL Generics zu sehen.

- CFG_MCTRL_SDEN = 1: aktiviert den SDRAM Controller
- CFG_MCTRL_SEPBUS = 0: verwendet gemeinsamen Bus mit PROM oder SRAM
- CFG_MCTRL_SDLSB = 1: fügt automatisch zusätzliche LSB Bits in die SDRAM Adressen ein, mit diesem Offset kann die unterschiedliche Adressierung zwischen FLASH und SDRAM kompensiert werden
- CFG_MCTRL_FAST = 0: deaktiviert schnelle Adressdecodierung, zwingend nötig für SDRAM Betrieb

Bemerkung zum SDRAM: Der SDRAM soll bei einem 32 Bit Zugriff zwei 16 Bit Zugriffe ausführen. Dies konnte jedoch noch nicht realisiert werden. Das *sdo.dqm* Signal würde zwar korrekt die oberen 16 und die unteren 16 Bits trennen und so zwei Zugriffe ermöglichen. Jedoch wird vom SDRAM Controller nur ein Write oder Read Kommando geschickt.

5.1.7 Top-Level & I/O Pads

Wie bereits erwähnt, ist die von AMBAdraw erstellte Schaltung nicht das Top-Level. Dieses wird nun selber erstellt.

5.1.7.1 Top-Level Es wird eine neue Library *leon_ toplevel* zum HDL Designer Projekt hinzugefügt. Dies wird mit Hilfe des *New Library...* Button durchgeführt. Der Library wird im Anschluss ein neues Blockschema hinzugefügt. Dies unter *New/Add...* und *Graphical View ⇒Block Diagram* .

Dem Blockschema kann nun ein neuer Block hinzugefügt werden (*Add Component*), nämlich der zuvor modifizierte *toplevel_ arch* Block aus der *ambarchitect* Library. Dieser Block weisst nun alle vorhin hinzugefügten Ports auf.

Das fertige Toplevel sieht wie folgt aus:

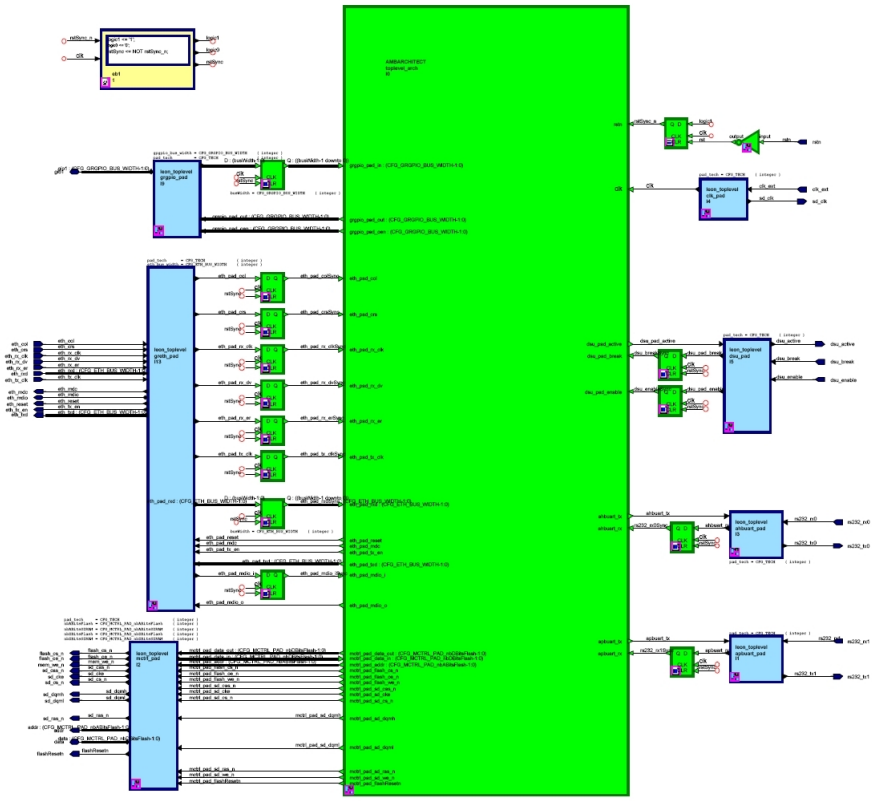


Abbildung 35: LEON Toplevel

Die restlichen Elemente des Toplevels werden nun beschrieben.

5.1.7.2 I/O Pads Die Signale und Busse, die von den Ports stammen, können nicht einfach so auf die FPGA übernommen werden. Jede FPGA Architektur handhabt diese unterschiedlich.

Gaisler bietet für diesen Zweck technologie-spezifische I/O Pads an. Diese werden in einer ersten Phase alle durch allgemeine Pads abstrahiert. Diese besitzen ein *tech* Generic. Dort kann die Ziel-Architektur der FPGA definiert werden. In diesem Fall mit der CFG_TECH Konstanten aus *config.vhd*. In der zweiten Phase werden automatisch die korrekten Pads für die entsprechende Technologie verwendet.

Es gibt folgende wichtige Pads:

- inpad: für einen Eingangsport hin zur FPGA
- outpad: für einen Ausgangsport von der FPGA weg
- iopad: für bi-direktionale Signale hin und von der FPGA
- clkpad: für den Systemclock, verwendet das Clocknetz der FPGA

Es gibt auch vektorbasierte Varianten, inpadv, outpadv und iopadv. Diese sind für Busse zu verwenden, die Busbreite lässt sich per Generic definieren.

Die Pads befinden sich allesamt in der *techmap* Library der GRLIB und müssen gegebenenfalls von dort aus importiert werden.

Um das Top-Level strukturiert zu halten, wird für jeden IP Core mit I/Os ein Unter-Blockschaltbild innerhalb des Top-Levels erstellt. Alle *_pad* Signale des IP Cores werden zum Unter-Blockschaltbild verbunden. Darin werden mit den oben aufgelisteten Pads, alle Signale und Busse an die verwendete Technologie, in diesem Fall *Spartan3E*, angepasst. Heraus kommen anschliessend die endgültigen I/Os der Schaltung, welche später bei dem Place and Route an die entsprechenden Pins der FPGA angeschlossen werden können. Folgende Abbildung zeigt ein solches Pad-Unter-Blockschaltbild auf:

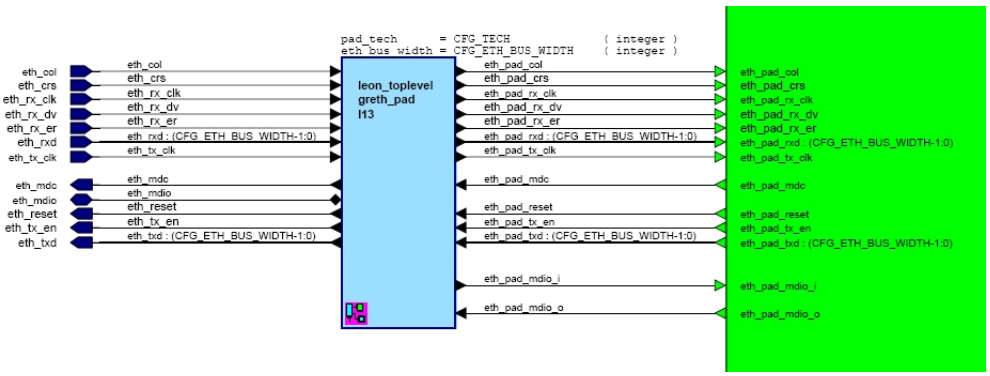


Abbildung 36: Pad-Unter-Blockschaltbild

Folgende Abbildungen zeigen Beispiele der Verwendung von Pads auf:

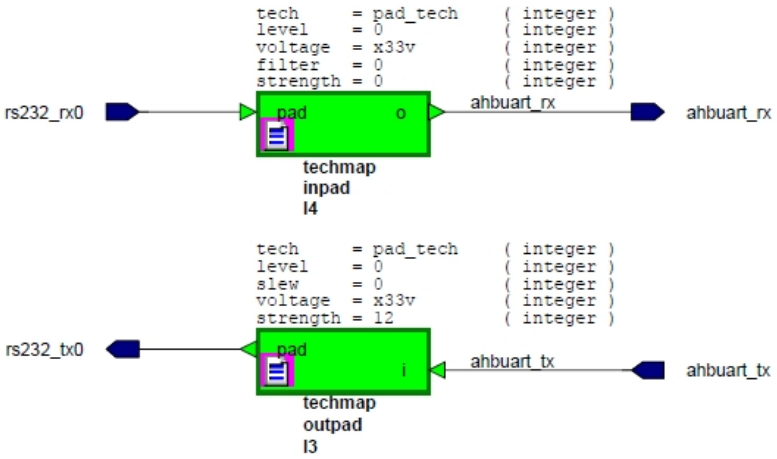


Abbildung 37: inpad & outpad

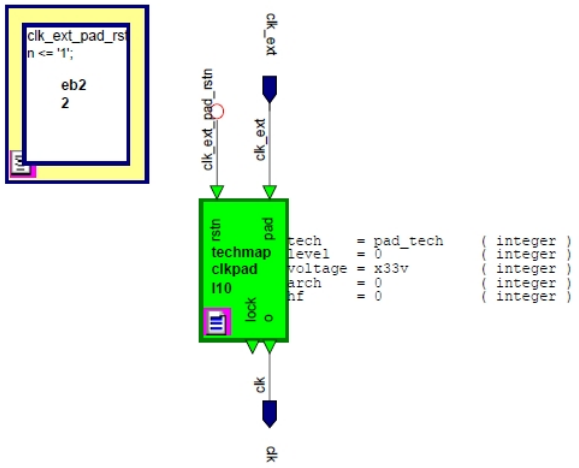


Abbildung 38: clkpad

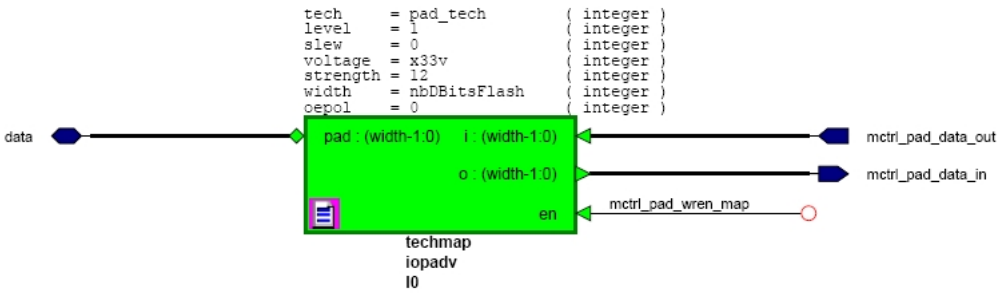


Abbildung 39: iopad

Das Enable Signal (en) eines iopads kann direkt mit einem Enable Signal eines IP Cores verbunden werden. Beim MCTRL Pad ist es aber nötig für den Databus ein Zwischen-Enable-Signal zu erstellen, welches die einzelnen Write Enables bzw. Befehle für SDRAM und FLASH vom Controller kombiniert. Der Code dazu ist in Beilage 15 zu sehen. Folgende Wahrheitstabelle zeigt die Funktionsweise eines iopads auf:

en	**Richtung**
0	i nach pad
1	pad nach o

Tabelle 6: Wahrheitstabelle iopad

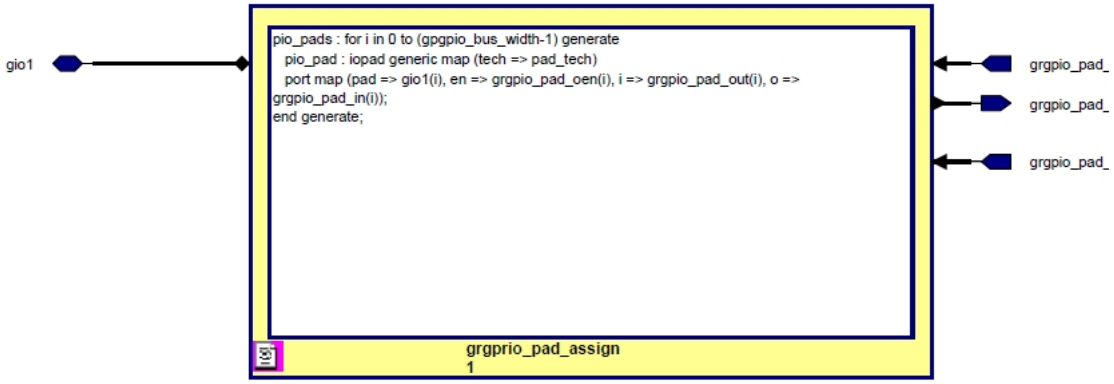


Abbildung 40: VHDL Code Pad

Das VHDL Code Pad für den General Purpose IP Core wurde direkt mit VHDL implementiert, da für jeden I/O des Busses ein eigenes Pad vorhanden sein muss. Dies wäre im Übrigen auch grafisch realisierbar gewesen indem man ein iopad zeichnet und dieses mit einem for-Loop-Block umschliesst.

In Beilage 15 sind alle VHDL Codes für die Pad-Blockschemas ersichtlich.

5.1.8 Synchronisation der Eingänge

Um ein allzeit korrektes Funktionieren sicherstellen zu können, müssen die Eingänge der Schaltung synchronisiert werden. Dies verhindert meta-stabile Zustände. Dazu werden D-Flip-Flops verwendet. Diese sind allesamt mit dem Systemtakt geschaltet und werden über einen ebenfalls synchronisierten Reset gecleart.

Folgende Abbildung zeigt die Synchronisation des Resets.

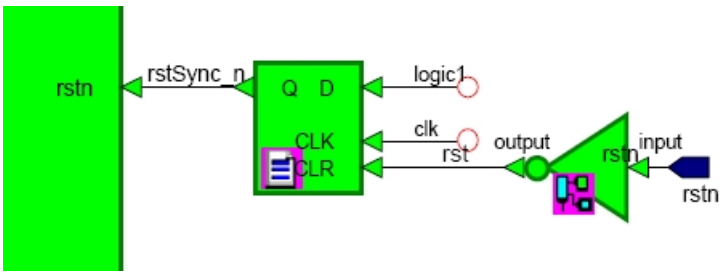


Abbildung 41: Reset Synchronisation

Das Reset Signal stammt von einem Button der aktiv tief arbeitet. Daher muss das Signal erst invertiert werden bevor es auf das Flip-Flop geführt werden kann. Der Eingang des Flip-Flops erhält eine logische '1', diese wird bei jedem Clock an den Ausgang weitergeleitet. Da der Reset der LEON Schaltung (rstn) aktiv tief ist, bedeutet diese '1', dass kein Reset vorliegt. Erst sobald der Button gedrückt wird, wird das Flip-Flop zurückgesetzt und es gibt eine '0' am Ausgang aus. Dies setzt anschliessend auch die LEON Schaltung zurück.

Für normale Eingangssignale wird untenstehende Konfiguration verwendet.

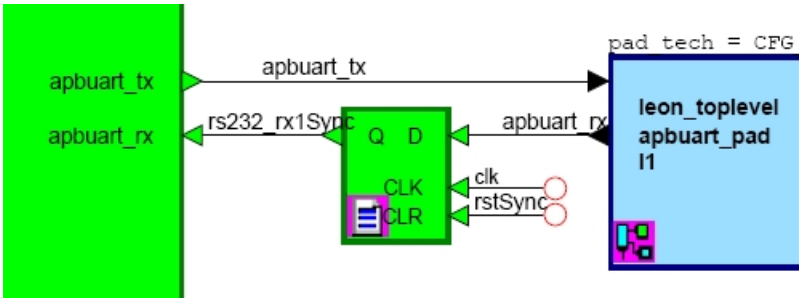


Abbildung 42: Signal Synchronisation

Das Flip-Flop erhält am Eingang das vom Pad her stammende Signal. Bei jedem Clock wird dieses an den Ausgang weitergeleitet. Es entsteht so zwar eine Verzögerung um eine Clockperiode, jedoch ist der Zustand des Signals am Ausgang stets stabil da genau der Wert während der steigenden Flanke des Clocks erfasst und weitergeleitet wurde.

Eingangsbusse werden mit einem leicht modifizierten D-Flip-Flop synchronisiert.

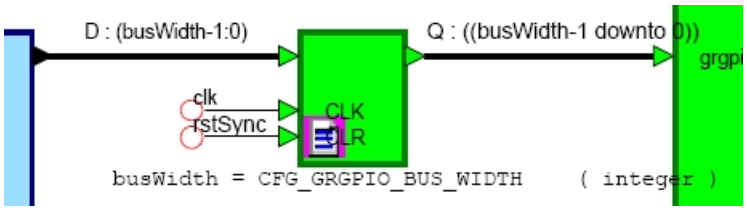


Abbildung 43: Bus Synchronisation

Das Flip-Flop hat die gleiche Funktionalität. Der Unterschied ist, dass sein Ein- und Ausgang jetzt ein Bus vom Typ *std_logic_vector* ist. Mit Hilfe eines Generics wird die Busbreite bestimmt. In HDL Designer ist dies eine einfach zu realisierende Lösung. In der FPGA wird dieses Bus-Flip-Flip in mehrere, der Busbreite entsprechend, einzelne Flip-Flops zerlegt.

Wichtig: Die Flip-Flops müssen alle hinter den Pads geschaltet werden. Die Pads stellen in der FPGA das letzte Element vor dem I/O Pin dar. Die Synchronisation muss zwischen Pad und dem Rest der Schaltung erfolgen.

5.1.9 Fehlende Blöcke importieren

Während dem Projekt ist es öfters vorgekommen, dass beim Generieren des VHDL Codes oder später während der Synthese oder des Mappings, Fehler aufgetreten sind, weil ein bestimmter Block gefehlt hat. Gaisler hat die GRLIB sehr verschachtelt aufgebaut. Es ist nicht immer direkt ersichtlich wie ein Block von einem anderen abhängt. Viele Blöcke verwenden intern weitere andere Blöcke und generieren aus denen neue Elemente. Z.B. besitzt der SRAM16 Block intern zwei SRAM Blöcke. Im Projekt müssen daher sowohl der SRAM16 Block als auch der SRAM Block vorhanden sein. Ansonsten wird ein Block nicht gefunden und das Projekt kann nicht fehlerfrei erstellt werden.

Dies gilt für das Generieren des Projekts, die Simulation und vor allem auch für die Synthese bzw. Implementation des Designs. Folgende wichtige Blöcke und Unterblöcke haben bei der Synthese der LEON Schaltung Fehler verursacht und müssen vorgängig bereits dem Projekt hinzugefügt werden um diese zu vermeiden:

- syncram_2p
- generic_syncram_2p
- virtex2_syncram_2p
- virtex2_syncram_dp
- grethc

Die fehlenden Blöcke können allesamt von der GRLIB ins HDL Projekt importiert werden. Welcher Library ein bestimmter Block angehört, kann durch eine einfache Suche nach der .vhd Datei des Blocks im GRLIB Verzeichnis gefunden werden. Im Pfad des Suchresultats ist ersichtlich wo sich der Block in der GRLIB befindet. Der Block muss nun in die gleiche Library innerhalb des *AMBArchitect.hdp* Projekts importiert werden.

Falls bei der Implementation des Designs erst ein Fehler auftritt, liegt das daran, dass das Synthese Programm, in diesem Fall Synplify Pro, für jeden nicht gefundenen Block eine Black Box generiert. Das Implementations- und Mapping Programm, hier Xilinx ISE, versucht nun diesen unbekannten Block in der FPGA zu platzieren. Das ist jedoch nicht möglich und dann wird eine Fehlermeldung ausgegeben wie die Folgende:

```
Checking expanded design ...
ERROR:NgdBuild:604 - logical block 'IO/I20/rf0/s1.dp.x1' with type
   'syncram_2p_leon_toplevel_toplevel_struct_1' could not be resolved. A pin
   name misspelling can cause this, a missing edif or ngc file, or the
   misspelling of a type name. Symbol
   'syncram_2p_leon_toplevel_toplevel_struct_1' is not supported in target
   'spartan3e'.
```

Abbildung 44: Fehlender Block - Fehlermeldung

Der Fehler, dass der Block nicht kompatibel zum Target spartan3e ist, scheint etwas verwirrend. Dies hat aber einen einfachen Grund. Normalerweise sind die Black Boxen, die vom Synthese Programm belassen werden, auf dem tiefsten Niveau in der Hierarchie des Designs. Also Elemente die direkt in der FPGA an bestimmten Orten platziert werden können. Z.B. einen Dual-Port RAM der ein Block RAM der FPGA einnehmen soll. Das Implementations- und Mapping Programm kann solche hardwareverbundenen Black Boxen identifizieren und korrekt in der FPGA platzieren.

Da es sich hier aber um einen fehlenden Block viel höher in der Hierarchie des Designs handelt, platziert das Syntheseprogramm die Black Box viel zu hoch und das Implementations- und Mapping Programm weiss nicht wo und wie dieser Block in der FPGA implementiert werden soll.

5.1.10 Simulation

Anschliessend kann die LEON Schaltung simuliert werden. Die Simulation ist ein wichtiger Schritt um die Funktionalität des Systems, vor dessen Implementation auf der Hardware, zu testen und allenfalls zu korrigieren.

Alle durchgeführten Simulationen sind im Kapitel Tests beschrieben.

5.1.11 Synthese und Place & Route

Nach erfolgreicher Simulation kann mit der Vorbereitung und der Implementierung der Schaltung für die FPGA begonnen werden. Dazu ist eine Synthese, ein Mapping und ein Place and Route nötig. Bei der Synthese wird der VHDL Code in eine Logik Schaltung mit Gattern, Flip Flops, usw. übersetzt. Dabei werden auch Speicher erkannt die z.B. in Block RAM der FPGA platziert werden könnten.

Beim Mapping werden den Lookup Tables (LUT) der FPGA ein Logikgatter zugewiesen. Die LUTs werden anhand ihrer Wahrheitstabelle konfiguriert. Jede FPGA Zelle besitzt neben dem LUT noch ein Flip Flop, diese können also so übernommen werden.

Beim Place and Route werden zuerst die Zellen gefüllt, also wo welches LUT und Flip Flop platziert wird. Dies entspricht dem Place Vorgang. Dabei können auch die erkannten Speicherelemente in verfügbare Block RAM gelegt werden. Beim Route werden alle Zellen untereinander gemäss der Schaltung verbunden. Ebenfalls wird der Clock auf dem Clocknetz verteilt. Auch werden alle I/Os der Schaltung den korrekten FPGA Pins zugewiesen.

5.1.11.1 Prepare for Synthesis Im HDL Designer wird im Design Manager die Library *leon_ toplevel* geöffnet und die Schaltung *toplevel* ausgewählt. Nun kann rechts *Prepare for Synthesis* gestartet werden. Dieser Vorgang generiert nochmals die ganze Schaltung und erstellt im Libraryverzeichnis im Unterordner */concat* ein .vhd File, welches die komplette Schaltung enthält. Diese Datei wird im Anschluss mit Hilfe eines Perl Scripts modifiziert um bestimmte, fehlerverursachende Library Includes zu entfernen. Für diesen Vorgang, sowie für den späteren Einsatz von Xilinx ISE, sind zudem einige HDL Designer Optionen zu ändern. Dazu öffnet man *Options* ⇒ *Main...* ⇒*User Variables*. Folgendes Fenster erscheint:

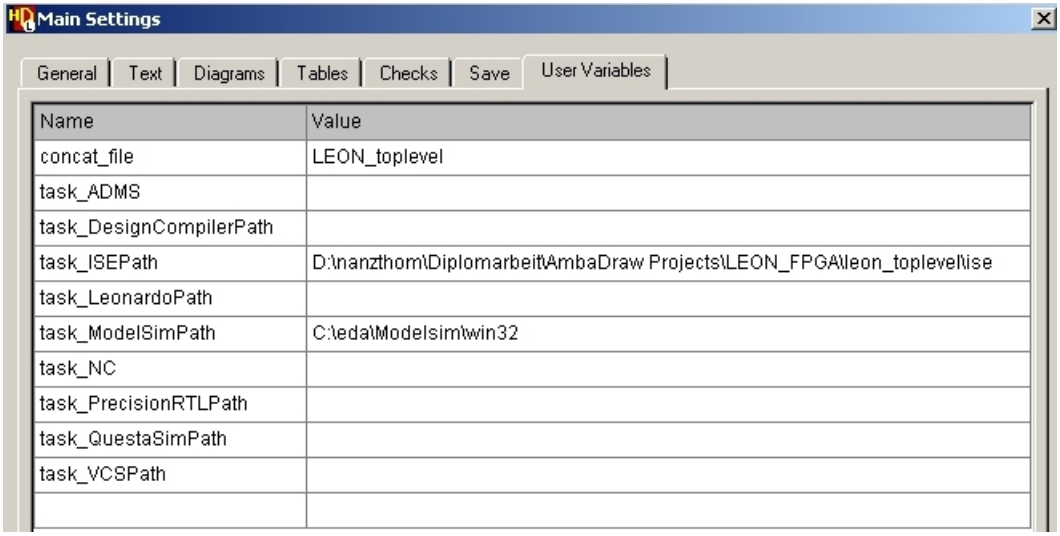


Abbildung 45: HDL Designer User Variables

Hier kann unter concat_file der Name des grossen .vhd Files definiert werden. Ebenfalls wird unter task_ISEPath der Pfad für das ISE Projekt konfiguriert. Hierbei ist zu empfehlen, einen Pfad ohne Leerzeichen zu wählen. Ansonsten, wie auch während dieser Arbeit, wird automatisch im root-Verzeichnis des Designs ein Projects Ordner angelegt. Darin wird das ISE Projekt gespeichert und auch ausgeführt.

Wichtig: bei Normalkonfiguration von HDL Designer müssen diese beiden User Variables bei jedem Wechsel zwischen Projekten neu angepasst werden.

5.1.11.2 Synplify Pro Beim oben erwähnten *Prepare for Synthesis* werden bei der LEON Schaltung nicht alle Libraries wie gewünscht vom Perl Script auskommentiert. Dies hat zur Folge, dass Xilinx ISE beim Synthetisieren eine Fehlermeldung ausgibt, z.B. bezüglich der Library *techmap*.

Daher wurde auf Synplify Pro ausgewichen, mit diesem Synthesetool lässt sich die Schaltung problemlos synthetisieren. Die Bedienung ist sehr simpel gehalten. Nach dem Prepare for Synthesis kann rechts *Synplify* gestartet werden.

Bemerkung: Ist kein solcher Link vorhanden, kann dieser mit *Rechtsklick* ⇒ *New Tool...* hinzugefügt werden. HDL Designer hat bereits vordefinierte TCL Plugins für Synplify. Es muss nur noch der Pfad und als Programm *Synplify Pro* gewählt werden. Für eine Schritt für Schritt Anleitung wird auf den Bericht von Herrn Valentini, Anhang C[6] verwiesen.

Synplify lädt nun ein neues Projekt, die Revisionen werden fortlaufend nummeriert. Es muss einzig die korrekte FPGA ausgewählt werden, dazu ist ein Doppelklick auf *rev_x* nötig. Es öffnet sich folgendes Fenster. Die Abbildung zeigt ausserdem die korrekte Auswahl der FPGA für dieses Projekt.

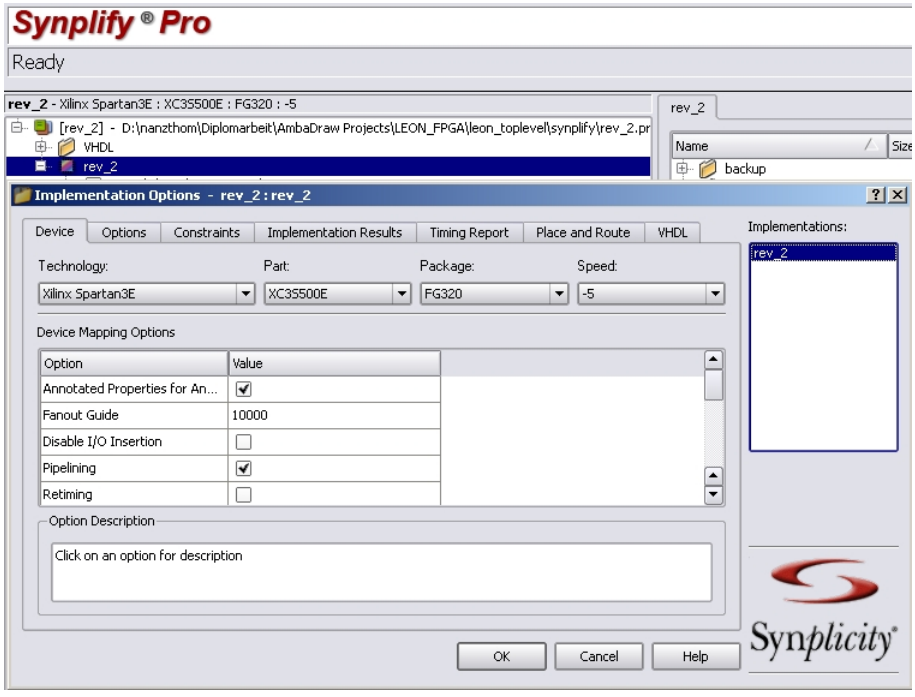


Abbildung 46: Synplify Pro - FPGA Einstellungen

[6] Valentini_TD_Bericht.pdf auf CD

Nun genügt ein Klick auf *Run* und die Synthese und ein Mappingvorgang starten. Dies kann relativ lange dauern. Das Programm erstellt daraufhin im Verzeichnis *...\LEON_FPGA\leon_toplevel\synplify* für die aktuelle Revision einen Ordner. Hier sind zwei Dateien wichtig:

- toplevel_struct.edf: Enthält die gesamte, synthetisierte Schaltung.
- synplicity.ucf: Enthält wichtige Bedingungen (Constraints) für die Schaltung, z.B. Timings oder Routelängen.

Diese beiden Dateien können nun in Xilinx ISE importiert werden. Mehr dazu im nächsten Kapitel.

Synplify bietet ausserdem die Möglichkeit nach der Synthese die Logikschaltung grafisch anzuschauen. Nach einem Klick auf *Technology View* (Gatter-Icon) öffnet sich ein neues Fenster. Hier ist nun die Logikschaltung hierarchisch dargestellt.

An dieser Stelle werden nur die Block RAMs für die LEON Register aufgezeigt. Diese stellen nun eine Black Box dar. Für das Syntheseprogramm sind diese Blöcke unbekannt. Da der Block RAM allerdings auf dem untersten Niveau ist, ist die Black Box gewünscht. In folgender Abbildung ist eine solche Block RAM Black Box zu sehen:

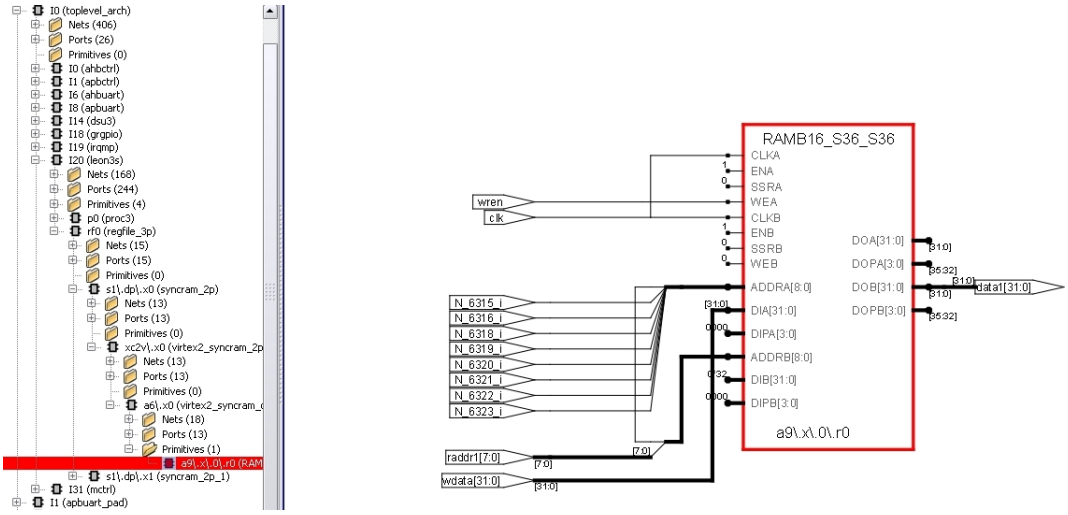


Abbildung 47: Synplify Pro - Block RAM Black Box

Mit dem *Technology View* ist es auch möglich alle I/Os und die Busbreiten einfach zu erkennen. Dies kann praktisch sein, falls eine Black Box selber definiert werden muss. Dies war allerdings während dem Projekt nicht nötig, Xilinx ISE hat alle Black Boxen erkannt.

5.1.11.3 Xilinx ISE Xilinx ISE wird nun direkt gestartet, nicht über das HDL Designer Script. Als erstes wird ein neues Projekt erstellt. Diesem wird als Source Type *EDIF* angegeben. Pfad und Projektname sind wie gewünscht zu setzen.

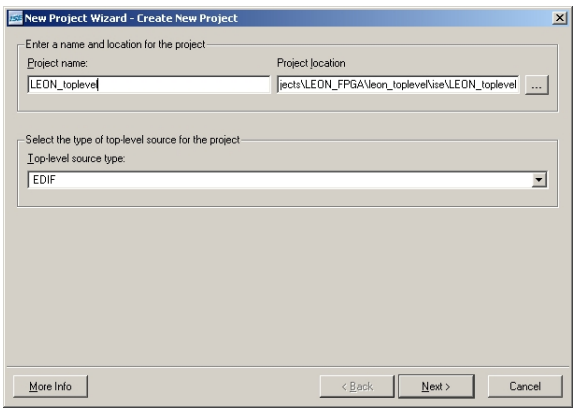


Abbildung 48: Xilinx ISE - Neues Projekt (1)

Als nächstes muss das .edf und das .ucf angegeben werden. Diese sind im *rev_x* Ordner des Synplify Projekts zu finden.

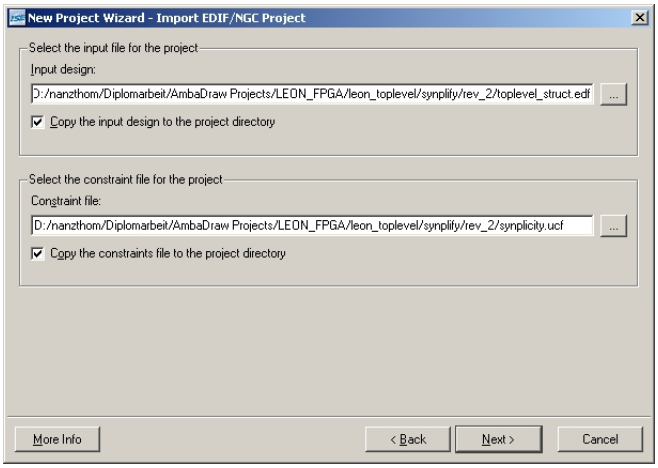


Abbildung 49: Xilinx ISE - Neues Projekt (2)

Im dritten Fenster werden die FPGA Settings eingestellt. Diese werden aber aus den Synplify Dateien korrekt ausgelesen. Dies ist dennoch zu verifizieren und mit *Next* zu bestätigen.

Nun werden bei den Quellen die zwei Dateien angezeigt, ebenfalls ist die korrekte FPGA zu sehen.

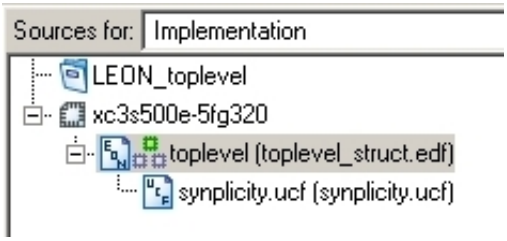


Abbildung 50: Xilinx ISE - Sources

5.1.11.3.1 Pinbelegung und .ucf Aktualisierung Im unteren Teil des ISE Fensters sind die Operationen und Befehle zu sehen. Wählt man bei den Sources die Hauptschaltung (.edf) aus, sind folgende Befehle ausführbar:

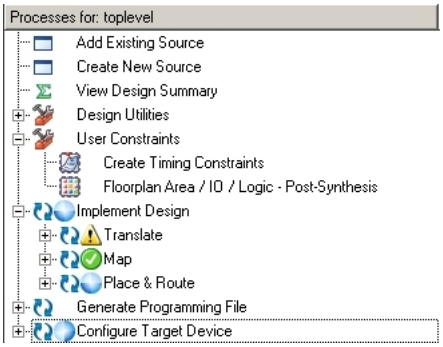


Abbildung 51: Xilinx ISE - Processes

Unter *User Constraints* ⇒ *Floorplan Area / IO / Logic - Post-Synthesis...* kann nun die Pinbelegung der FPGA angepasst werden. Dazu wird Xilinx PACE gestartet. Mit Hilfe dieses Tools lassen sich alle in der Schaltung vorhandenen I/Os an einen bestimmten Pin der FPGA anschliessen. Es muss für jedes Signal nur unter *Loc* der Pin eingetippt werden.

Wie welches Signal angeschlossen werden muss, ist im Schema[7] zum FPGA_EBS_V2.0 Board zu sehen. Das fertige .ucf ist in Beilage 16 angefügt.

[7] Pfad P:\PCB\Produit\Logical\FPGA-EBS\FPGA_EBS_V2.0\schematics.pdf oder CD zur Diplomarbeit

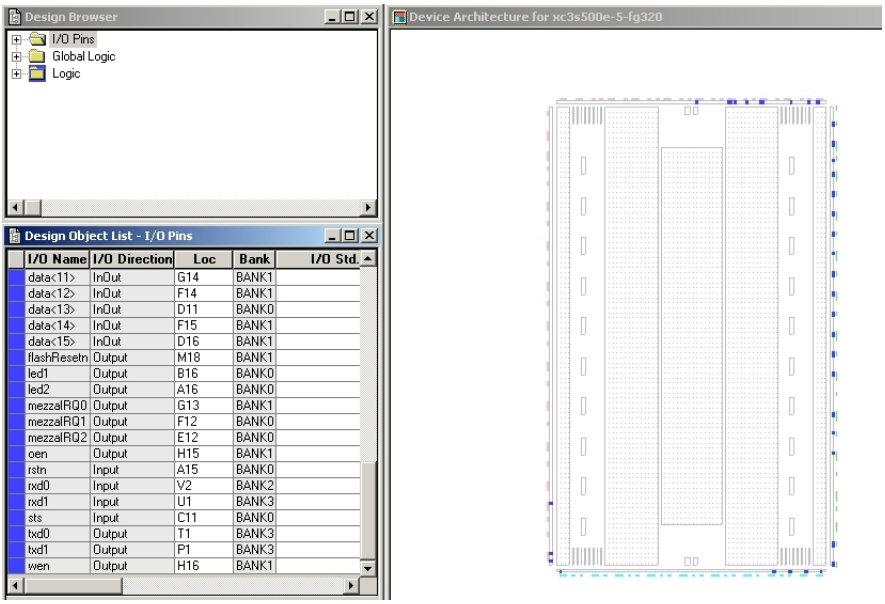


Abbildung 52: Xilinx ISE - PACE

Manche Signale erfordern einen bestimmten Abschluss (Termination), z.B. einen Pullup Widerstand. Die Pads in der FPGA können so entsprechend konfiguriert werden.

Folgende wichtige Punkte sind zu beachten:

- Signal rstn mit Pullup Abschluss (Button A15)
- Signal dsu_enable mit Pullup Abschluss (Button D3)
- Signal dsu_active an LED A16
- Adressen für FLASH und SDRAM können 1:1 gemappt werden, A0 an MEZ_A00_NLB

Wichtig: Beim ersten Start muss die Pinbelegung von Grund auf erstellt werden. PACE fügt alle Mappings dem .ucf, welches von Synplify erstellt wurde, hinzu. Damit die Pinbelegung nicht bei jeder neuen Revision eingegeben werden muss, empfiehlt es sich die .ucf Datei aus dem ISE Projekt Ordner zu sichern.

Allerdings ist es nicht garantiert, dass Synplify bei jeder Revision die gleichen Constraints in sein synplicity.ucf schreibt. Da diese Constraints allerdings wichtig sind, muss bei jeder neuen Revision die synplicity.ucf Datei im *rev_x* Verzeichnis modifiziert werden um die Pinbelegung beizubehalten. Dazu werden von Hand alle Zeilen, welche von PACE dem alten .ucf im ISE Projektverzeichnis hinzugefügt wurden, in das neue synplicity.ucf im *rev_x+1* Verzeichnis kopiert. Diese neue synplicity.ucf Datei kann zusammen mit der neuen .edf Datei nun ins ISE Projekt importiert werden. Dabei müssen die alten Dateien überschrieben werden.

5.1.12 Download auf FPGA

Nun sind alle nötigen Schritte durchgeführt, die Schaltung kann auf das FPGA_EBS_V2.0 Board geladen werden. Dazu wird der Prozess *Configure Target Device* gestartet. ISE führt nun ein Mapping und ein Place & Route durch. Im Anschluss wird ein Programming File generiert, welches mit iMPACT über das JTAG Interface auf die FPGA geladen werden kann. iMPACT wird automatisch gestartet sobald ISE alle anderen Prozesse beendet hat.

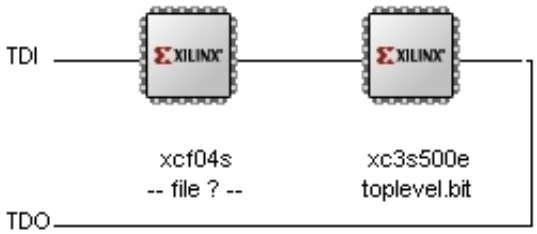


Abbildung 53: Xilinx ISE - iMPACT

Auf obenstehender Abbildung ist die gefundene JTAG Kette zu sehen. Der linke Chip ist das EEPROM. In dieses könnte die Schaltung gespeichert werden, welche bei jedem Aufstarten des Boards in die FPGA geladen werden soll. Das wurde aber für diese Schaltung noch nicht verwendet. Im Bericht von Herrn Valentini ist in Anhang C aufgezeigt wie das EEPROM programmiert wird.

Der rechte Chip ist die FPGA. Diesem Chip muss nun das Programming File (.bit) zugewiesen werden. Dazu wird der Chip doppelt angeklickt und es wird das *toplevel.bit* File gewählt. Mit einem *Rechtsklick* ⇒ *Program* wird die FPGA nun geschrieben. Nach erfolgreicher Programmation leuchtet die DONE LED auf dem Board auf. Kann die FPGA mehrmals hintereinander nicht korrekt programmiert werden, hilft meistens ein kurzes Entfernen der Spannungsversorgung.

5.1.13 FPGA limitiert Grösse der LEON Schaltung

Die eingesetzte FPGA weisst leider eine zu kleine Kapazität auf für die geplante LEON Schaltung. Bei der Synthese mit Synplify wurde festgestellt, dass die eingesetzten LUTs mehr als nur im kritischen Bereich sind. Über 80% LUT Belegung gilt als kritisch da dies für ein sauberes Place and Route keine gute Voraussetzung ist. Bei zu hoher Belegung kann es vorkommen, dass gewisse Leitungen bzw. Signale nicht mehr innerhalb der vorgeschriebenen Bedingungen (Constraints) geroutet werden können.

Die LEON Schaltung wurde daraufhin auf Minimalbedarf getrimmt, das heisst es wurden beispielsweise die beiden Caches des LEON3 deaktiviert, alle verwendeten Buffer (aller Cores) auf einen Minimumwert gesetzt usw.

Der Ethernet Controller bleibt dennoch der Hauptverantwortliche für den Kapazitätsbedarf. Er realisiert seine zwei RX- und TX Buffer mit verteiltem Speicher (distributed RAM). Das heisst er verwendet sehr viele LUTs und Flip Flops um seine Buffer zu erstellen. Diese sind in folgender Abbildung zu sehen:

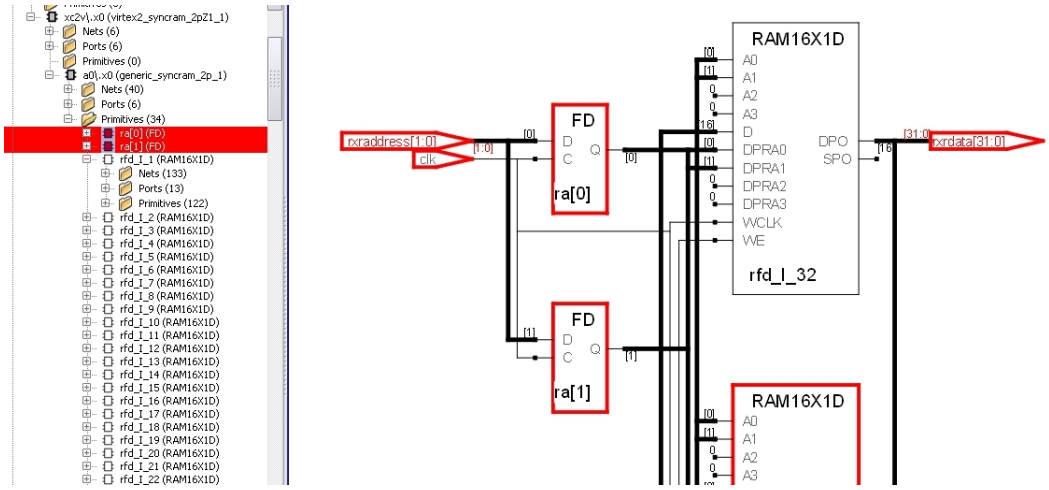


Abbildung 54: GRETH - Distributed RAM

Pro FIFO werden sehr viele Black Boxen erstellt. Xilinx ISE ist nicht in der Lage diese auf Block RAMs zu setzen und verwendet stattdessen den Bereich der FPGA, der eigentlich der Schaltung zukommen sollte.

Folgende Tabelle zeigt die LUT Auslastung mit und ohne GRETH Core (Angaben aus Synplify Log):

GRETH	LUT Auslastung [%]
ein	ca. 105%
aus	ca. 91%

Tabelle 7: LUT Auslastung - mit und ohne GRETH

Je nach Revision der Schaltung ändert sich die Auslastung minimal.

Bereits ohne Ethernet Controller ist die FPGA kritisch ausgelastet, jedoch noch realisierbar. Mit GRETH kann Xilinx ISE die Schaltung nicht für die FPGA vorbereiten.

Als temporäre Lösung wurde der Ethernet Controller deaktiviert. Dazu wurde in der Schaltung um den IP Core und dessen Signal Routes Block ein IF-Frame gelegt, welches auf eine, in der *config.vhd* definierte, Konstante USE_ETH reagiert. Dann wurde ein zusätzlicher Embedded Block erstellt und ebenfalls mit einem IF-Frame versehen. Diese Bedingung reagiert auf den invertierten Wert der Konstante. So wird der Embedded Block aktiv, falls GRETH nicht verwendet werden soll. Im Embedded Block werden die Eingangssignale von GRETH auf Don't Care ('-') gesetzt, so dass sie nicht in der Luft hängen.

Als mögliche Lösung käme ein Block RAM Forcieren in Frage. Diese Lösung wurde im Rahmen der Diplomarbeit allerdings nicht versucht. Aber es könnte möglich sein, den Ethernet Controller so umzuändern, dass Block RAM statt distributed RAM verwendet wird. So würde ein grosser Teil der LUT Auslastung auf die Block RAM Auslastung übergehen. Die Block RAM Auslastung beträgt ca. 30% bei der LEON Schaltung, dort wäre also noch Kapazität vorhanden.

Diese Änderung halt auch Folgen auf die Demoapplikation. Statt über Ethernet wird die Kommunikation nun über RS-232 durchgeführt.

5.2 USB Controller Core

Der USB Controller Core wurde nicht im Rahmen dieser Diplomarbeit realisiert. Die LEON Schaltung ist ohne USB Controller Core bereits so gross, dass die verbaute FPGA an die Grenzen ihrer Kapazität stösst.

5.3 FLASH Programmation

Nun wird die Schaltung zur Programmation des FLASH Speichers gemäss dem im Spezifkationskapitel beschriebenen Blockschaltbild und Übertragungsprotokolls implementiert. Dazu wird in HDL Designer ein neues Projekt namens *Flash_Prog* (*Flash_Prog.hdp*) erstellt. In einem nächsten Schritt wird eine neue Library *Flash_Prog_lib* erstellt. Nun kann in dieser Library ein neues Blockschaltbild *flash_pr_toplevel* hinzugefügt werden. Dies unter *New/Add...* und *Graphical View* ⇒*Block Diagram* .

In dieser Schaltung wird das Blockschaltbild bestehend aus RS-232 Interface, Control Extractor (Management) und Zustandsmaschine realisiert und mit den nötigen Signalen bzw. Bussen verbunden. Folgende Abbildung zeigt den Inhalt des Toplevels auf:

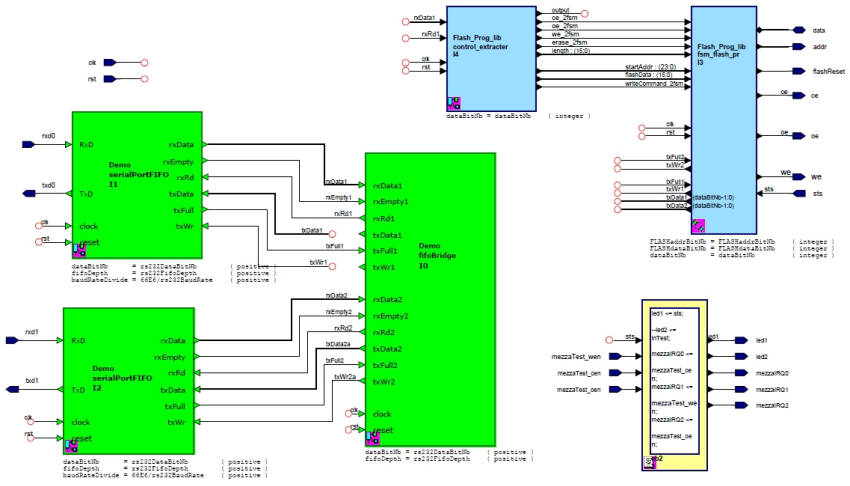


Abbildung 55: FLASH Programmation - *flash_pr_toplevel*

Wie zu sehen ist, sind alle Blöcke mit den gleichen 66MHz Clock (clk) und Reset (rst) Signalen synchron geschaltet. Dies ist zwingend nötig damit die Kommunikation auf allen Ebenen zeitsynchron durchgeführt wird. Konkret heisst das, dass beispielsweise das Versenden und das Abtasten von bestimmten Kontrollsignalen, z.B. bei der Ankunft eines neuen Datenpakets auf der seriellen Schnittstelle, genau zeitgleich ausgeführt werden. Die Blöcke arbeiten so synchron zusammen. Der gelbe Embedded Block verbindet Signale wie cen, oen, wen und das STS Signal zu anderen Pins bzw. LEDs auf dem FPGA_EBS_V2.0 Board. Da die Ausgänge cen, oen und wen nicht direkt gelesen werden können, werden sie auf dem Toplevel auf die mezzaTest Eingänge zurück geführt (siehe untenstehende Abbildung). Die mezzaIRQ Signale werden ebenfalls nach aussen geführt. Der FLASH Speicher ist zudem deutlich schneller als das RS-232 Interface, es muss kein Zusätzlicher Buffer implementiert werden. Das Toplevel ist in voller Grösse in Beilage 17 angefügt.

Da gewisse Ausgänge wie z.B. Output Enable oder Chip Enable aktiv tief sein müssen, werden in einer weiteren Schaltung letztendlich noch Inverter für diese Signale hinzugefügt. Im Endeffekt ist dies das Toplevel des Projekts. Dazu wird wiederum eine neue Library namens *Flash_Prog_Toplevel* erstellt. Darin wird, wie oben bereits beschrieben, ein neues Blockschaltbild (Block Diagram) *toplevel* hinzugefügt. In dieser Schaltung kann nun *flash_pr_toplevel* als Block hinzugefügt werden. Für die Signale ce, oe, we, flashReset und rst werden Inverter hinzugefügt, da diese Signale allesamt aktiv tief sind. Der Vorteil die Signale erst auf Stufe des Toplevels zu invertieren liegt darin, dass die Signale innerhalb der Unterschaltungen alle als aktiv hoch betrachtet werden können. Dies erleichtert die Entwicklung einer Schaltung.

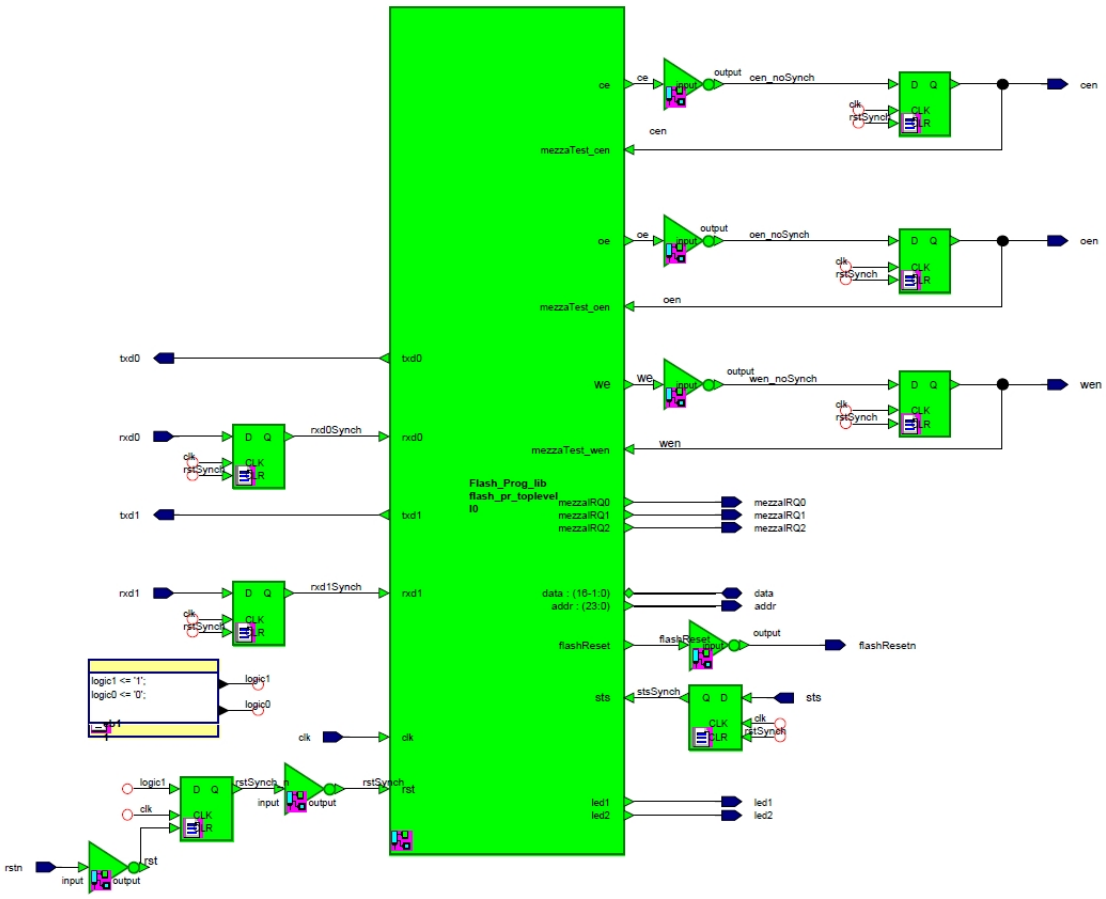


Abbildung 56: FLASH Programmation - *toplevel* & Inverter

Ebenfalls werden auf diesem Niveau alle Eingänge mit D-Flip-Flops synchronisiert, so wie dies auch bei der LEON Schaltung der Fall war. Dies ist entscheidend um meta-stabile Zustände zu vermeiden. Die D-Flip-Flops werden allesamt mit dem Systemclock getaktet. Gecleart werden sie über ein ebenfalls synchronisiertes Reset Signal. Der Reset wird synchronisiert, da er von einem Button stammt. Dieser Button ist zudem aktiv tief und muss vor dem Flip-Flop selber invertiert werden. Zusätzlich wurden auch die Ausgänge cen, oen und wen synchronisiert. Die mezzaIRQ Signale können mit der Mezza-Debug Erweiterungskarte an einfach erreichbare Pins zur Messung mit dem Oszilloskop weitergeleitet werden. Ausserdem stehen direkt alle Adress- und Datenbits auf der Erweiterungskarte zur Verfügung da der FLASH mit den gleichen Linien verbunden ist wie das Mezzanine Interface.

Nun werden alle Blöcke im Detail erklärt.

5.3.1 RS-232 Interface

Das RS-232 Interface besteht aus zwei Hauptelementen. Diese wurden von der HES-SO Wallis zur Verfügung gestellt. Auf deren Funktionalität wird daher nicht im Detail eingegangen. Diese Blöcke wurden aus dem *ELN_board* Projekt importiert. In dem Projekt waren die Blöcke in einer Library namens *Demo* implementiert. Um zu verhindern, dass alle Namen, Generics und sonstige Parameter neu definiert werden müssen, wurde im Projekt zur FLASH Programmation eine neue Library namens *Demo* hinzugefügt und darin können nun alle Elemente der *Demo* Library aus dem zur Verfügung gestellten Projekt importiert werden. Verwendet werden allerdings nur folgende Blöcke, dies ist auch in untenstehender Abbildung zu erkennen:

- **serialPortFIFO:** Dieser Block stellt einen RS-232 Port dar und wandelt beim Empfang die seriellen Signale (RxD) in einen parallelen Bus bzw. in ein Datenpaket (rxData) um. Analog dazu kann beim Versenden von Daten ein Datenpaket (txData) serialisiert (TxD) werden.
  Zugleich bietet dieser Block auch einen Empfangs- und einen Versende First-In-First-Out (FIFO) Buffer an. So können Geschwindigkeitsunterschiede zwischen dem langsameren RS-232 und anderen, schnelleren Blöcken kompensiert werden.
  Die Grösse der Datenpakete, der FIFO Buffer und der Baud-Rate können per Generics konfiguriert werden. Der Block wird zwei Mal in die Schaltung aufgenommen, so sind beide verbauten RS-232 Ports ansprechbar.
- **fifoBridge:** Wie der Name schon sagt, stellt dieser Block die Brücke zwischen den beiden Ports dar. Er kontrolliert die von den FIFOs herkommenden Kontrollsignale rxEmpty und txFull und handelt entsprechend. Wenn rxEmpty auf '0' ist, ist der FIFO nicht leer und es können Daten auf dem rxData Signal bezogen werden. Wird dies gemacht, wird das rxRd Signal für eine Clock-Periode aktiv gesetzt um ein Lesen der Daten zu signalisieren. Ist rxEmpty auf '1', ist der FIFO leer und es sind entsprechend keine Daten verfügbar.
  Analog dazu können, falls txFull auf '0' ist, Daten über txData versendet werden. Dabei wird das Signal txWr für eine Clock-Periode aktiviert. Ist txFull auf '1', ist der FIFO Buffer voll und es können keine weiteren Daten verschickt werden.
  Der Block bietet diese Funktionalität für zwei Ports an und bildet eine Brücke zwischen diesen.

Intern bestehen diese beiden Haupt-Blöcke aus weiteren Unterblöcken. Auf diese wird allerdings nicht weiter eingegangen.

Die Konfiguration der einzelnen Generics lautet wie folgt:

Block	Generic	Zugewiesene Konstante	Wert
serialPortFIFO	dataBitNb	rs232DataBitNb	8
	fifoDepth	rs2323FifoDepth	8
	baudRateDivide	66E6/rs232BaudRate	66E6/9600
fifoBridge	dataBitNb	rs232DataBitNb	8
	fifoDepth	rs2323FifoDepth	8

Tabelle 8: FLASH Programmation - RS-232 Generics

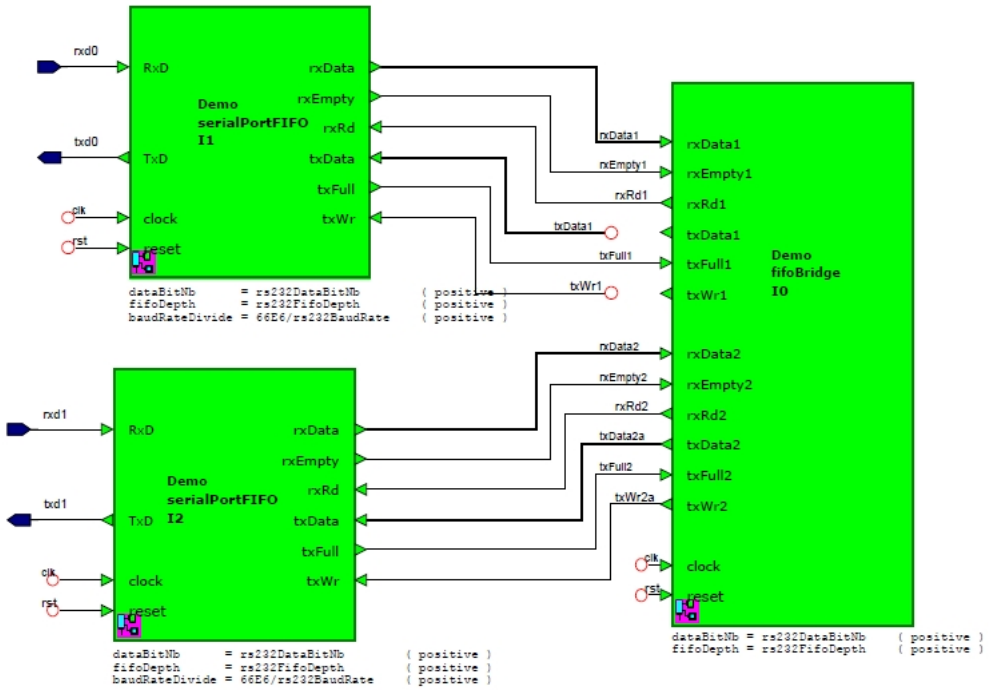


Abbildung 57: FLASH Programmation - RS-232 Interface

5.3.2 Control Extractor

Dieser Block wurde selber erstellt und hat als Aufgabe die Analyse der ankommenden Datenpakete. Er filtert Kontrollwerte von Datenwerten und liefert am Ausgang alle wichtigen Informationen für die Zustandsmaschine.

5.3.2.1 Ablauf Der Ablauf des Control Extractors wird anhand des folgenden Flow Charts dargestellt. Da dieser Block in VHDL implementiert ist, laufen alle Vorgänge parallel ab. Das Flow Chart dient nur zur übersichtlichen Darstellung des Aufgabenbereichs dieses Blocks.

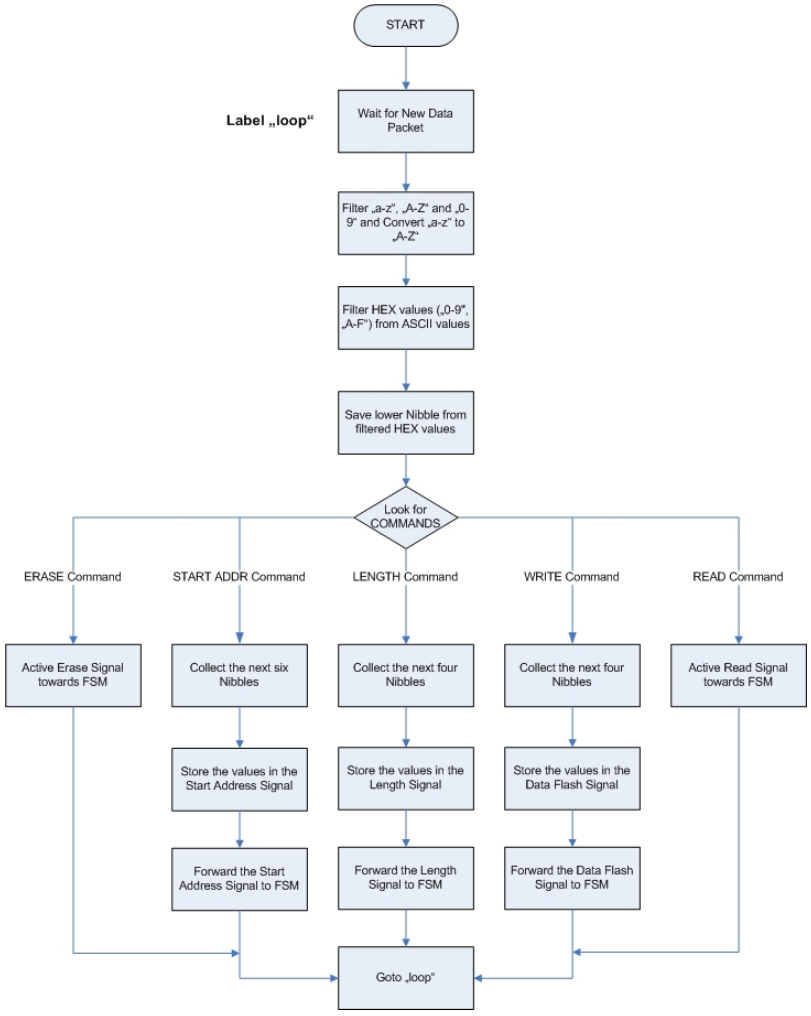


Abbildung 58: FLASH Programmation - Control Extractor - Flow Chart

Wie zu sehen ist, werden ankommende Datenpakete auf gültige Werte gefiltert. Gültige Werte können die ASCII Werte für Befehle sein, wie sie im Spezifikationskapitel beschrieben sind, oder auch Daten im HEX Format. Zugleich werden alle empfangenen Daten gross- und kleinschreibungsunabhängig gemacht, indem alle Buchstaben in Grossbuchstaben umgewandelt werden. Von allen gültigen ASCII Werten werden die niederwertigeren 4 Bits (Nibble) so gespeichert, dass der Wert des Nibbles genau dem HEX Wert (0-9, A-F) entspricht, der übermittelt wurde. Daraufhin wird kontrolliert ob es sich bei dem übermittelten Paket um einen Befehl gehandelt hat. Falls ja wird entsprechend gehandelt. Dabei wird im Falle eines START ADDR, LENGTH oder WRITE Befehls die folgenden Nibbles, für eine bestimmte Dauer, gesammelt und in einem 16 oder 24 Bit Signal zusammengefügt. In allen Fällen wird die hinten angeschaltete FSM über deren neue Aufgabe informiert.

Die Sequenz der Befehle ist fix definiert (siehe Spezifikationskapitel) und das Perl Script stellt diese Abfolge sicher.

5.3.2.2 Ein- und Ausgänge

Nun folgt ein Überblick über die Ein- und Ausgänge dieses Blocks.

Signal/Bus	Zweck
rxData1	Empfang eines 8 Bit Datenpakets. Herkommend vom I1 serialPortFIFO Block.
rxRd1	Signalisiert ein empfangsbereites Datenpaket auf rxData1. Herkommend vom I0 fifoBridge Block.

Tabelle 10: FLASH Programmation - Control Extractor - Eingänge

Signal/Bus	Zweck
output	Testausgangssignal, nicht weiter verwendet.
ce_2fsm	Chip Enable Signal für die FSM. Allerdings nicht implementiert.
oe_2fsm	Output Enable Signal für die FSM (Register). Zeigt einen Lesevorgang an. Die FSM arbeitet anschliessend ihren Lese-Ablauf ab.
we_2fsm	Write Enable Signal für die FSM (Register). Startet einen Schreibvorgang sobald Daten vorhanden sind. Die FSM steuert anschliessend den FLASH im Schreibmodus an.
erase_2fsm	Erase Signal für die FSM. Zeigt einen Löschvorgang an (Register). Die FSM arbeitet anschliessend ihren Lösch-Ablauf ab.
length	Mit diesem Signal wird die Länge der zu übertragenden Daten hin oder vom FLASH übermittelt (Register). Dient als Kontrollmittel für die Dauer eines Schreib- oder Lesezugriffs.
startAddr	Mit diesem Signal wird die Start Adresse des Schreib- oder Lesevorgangs übermittelt (Register).
flashData	Mit diesem Signal werden die zu schreibenden Daten übermittelt (Register). Es ist ein 16 Bit Signal da der FLASH Speicher ebenfalls 16 Datenleitungen besitzt.
writeCommand_2fsm	Write Command Signal für die FSM (Register). Zeigt einen Schreibvorgang an. Die FSM arbeitet anschliessend ihren Schreib-Ablauf ab.

Tabelle 12: FLASH Programmation - Control Extractor - Ausgänge

Diese Signale sind auf untenstehender Darstellung ebenfalls zu sehen. Der Block besitzt ein Generic *dataBitNb*. Damit lässt sich wiederum die Grösse eines Datenpakets definieren. Um kohärent mit den restlichen Blöcken zu sein, wurde dieses Generic mit der selben Konstante konfiguriert. Somit handelt es sich um 8 Bit Datenpakete.

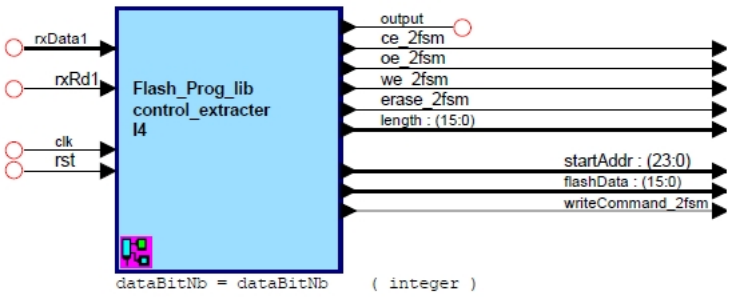


Abbildung 59: FLASH Programmation - Control Extractor Block

5.3.2.3 Interner Aufbau und Signale Intern ist dieser Block mit einem Unterblockschaltbild realisiert. Er besitzt neben den oben erwähten Ein- und Ausgängen inkl. entsprechenden Ports auch zahlreiche interne Signale. Mit Hilfe eines gelben Embedded-Blocks wird die Logic in VHDL Code implementiert. Mehr dazu in Kürze. Folgende Abbildung zeigt den internen Aufbau des Control Extractors.

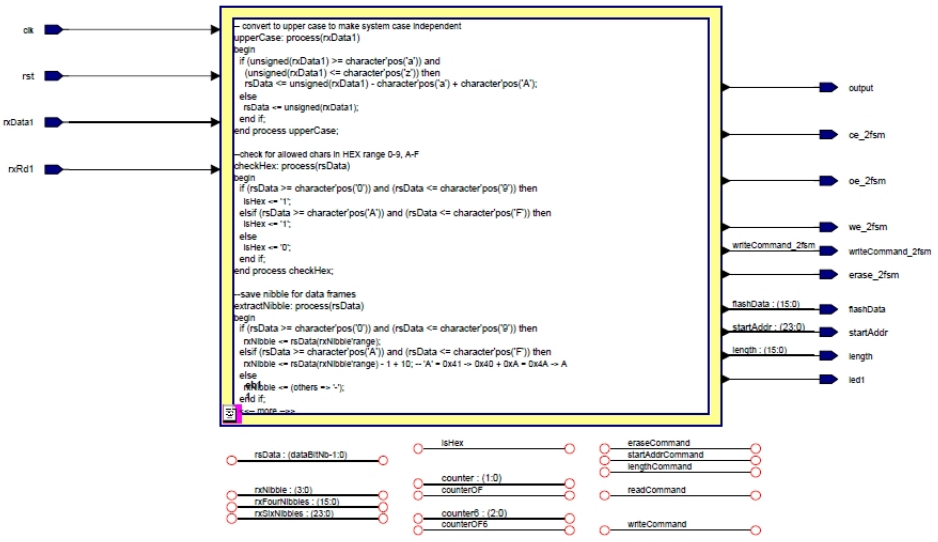


Abbildung 60: FLASH Programmation - Control Extractor Intern

Die internen Zwischensignale haben folgenden Verwendungszweck:

Signal/Bus	Zweck
rsData	Empfangene Daten auf rxData1 werden in Grossbuchstaben umgewandelt und auf diesem Signal für andere Prozesse gespeichert.
rxNibble	Speichert die vier niederwertigeren Bits des ASCII Wertes so, dass der Wert dem übertragenen HEX Wert entspricht.
rxFourNibbles	Auf diesem Signal werden die vier zuletzt empfangenen Nibbles zusammengefasst.
rxSixNibbles	Auf diesem Signal werden die sechs zuletzt empfangenen Nibbles zusammengefasst.
isHex	Dieses Signal ist aktiv, falls es sich um einen gültigen HEX Wert handelt. Dient als Kontrollsignal für viele Prozesse.
counter	Sich wiederholender Zähler von 3 nach 0, verantwortlich für die korrekte Zusammenstellung von rxFourNibbles.
counterOF	Zeigt einen Overflow des counter Zählers auf. Daraufhin werden die Signale length oder flashData aktualisiert.
counter6	Sich wiederholender Zähler von 5 nach 0, verantwortlich für die korrekte Zusammenstellung von rxSixNibbles.
counterOF6	Zeigt einen Overflow des counter6 Zählers auf. Daraufhin wird das Signal startAddr aktualisiert.
eraseCommand	Aktiv bei ERASE Command, ansonsten auf '0'. Kontrollsignal für andere Prozesse.
startAddrCommand	Aktiv bei START ADDR Command, ansonsten auf '0'. Kontrollsignal für andere Prozesse.
lengthCommand	Aktiv bei LENGTH Command, ansonsten auf '0'. Kontrollsignal für andere Prozesse.
readCommand	Aktiv bei READ Command, ansonsten auf '0'. Kontrollsignal für andere Prozesse.
writeCommand	Aktiv bei WRITE Command, ansonsten auf '0'. Kontrollsignal für andere Prozesse.

Tabelle 14: FLASH Programmation - Control Extractor - Interne Signale

5.3.2.4 VHDL Implementierung In Beilage 18 ist der VHDL Code zur Architektur des Control Extractors beigefügt. Darin sind alle oben erklärten Ein- und Ausgänge, interne Signale und natürlich der Ablauf implementiert.

Die implementierten Prozesse lauten wie folgt:

- **upperCase:** filtert die gültigen ASCII Werte und wandelt die empfangenen Werte in Grossbuchstaben um.
- **checkHex:** setzt das Signal isHex aktiv, falls es sich bei dem empfangenen Wert um einen HEX Wert handelt.
- **extractNibble:** speichert die vier niederwertigsten Bits des ASCII Wertes, siehe Kommentar für korrekte HEX-Umwandlung.
- **storeControls:** filtert auf Befehls-ASCII Werte und setzt die Kontrollsignale entsprechend.
- **nibbleCounter:** Zählprozess des 2 Bit Counters counter von 3-0. Eingesetzt für LENGTH und WRITE Befehle.
- **nibbleCounterOF:** setzt counterOF aktiv sobald counter überläuft.
- **nibbleCounter6:** Zählprozess des 3 Bit Counters counter6 von 5-0. Eingesetzt für START ADDR Befehl.
- **nibbleCounterOF6:** setzt counterOF6 aktiv sobald counter6 überläuft.
- **setFourNibbles:** kombiniert die vier zuletzt empfangenen Nibbles zu einem 16 Bit Signal. Eingesetzt für LENGTH und WRITE Befehle.
- **setSixNibbles:** kombiniert die sechs zuletzt empfangenen Nibbles zu einem 24 Bit Signal. Eingesetzt für START ADDR Befehl.
- **setOutputs:** aktualisiert je nach aktuellem Befehl das length oder flashData Signal mit dem neuen Wert von rxFourNibbles. Bei einem WRITE Befehl wird der FSM zudem ein neuer Schreibvorgang angezeigt.
- **setStartAddr:** aktualisiert bei START ADDR Befehl das startAddr Signal mit dem neuen Wert von rxSixNibbles.
- **setEraseSignals:** setzt bei aktuellem ERASE Befehl das Löschsignal für die FSM.
- **setReadSignals:** setzt bei aktuellem READ Befehl das Lesesignal für die FSM.

Wie in der Tabelle der Ausgänge dieses Blocks zu sehen ist, wird die Länge der Übermittlung, die Startadresse sowie die Daten mit Hilfe von Registern an die FSM übermittelt. Auch die Steuersignale, wie z.B. oe_2fsm, sind wie Register aufgebaut. Damit diese Register bei der Synthese als Flip-Flops interpretiert und umgesetzt werden, sind alle Prozesse, welche mit diesen Registern arbeiten, synchron mit der steigenden Flanke des Clocks oder mit einer if-else Bedingung implementiert. Im Falle des Clocks wird bei einem Reset ein Startwert festgelegt und die Speicherung des neuen Werts wird nur bei jedem neuen Clock durchgeführt. Beim if-else Vorgehen verfügt die else Bedingung einen Default Wert für das Signal. So können unerwünschte Latches bei der Synthese verhindert werden. Ausserdem arbeitet das gesamte System synchron mit dem Clock.

Auf eine detaillierte Codebeschreibung wird an dieser Stelle verzichtet. Es wird auf die Kommentare und die oben aufgeführten Erläuterungen verwiesen.

5.3.3 FSM Zustandsmaschine

Die Zustandsmaschine steuert anhand der vom Control Extractor erhaltenen Informationen, dies können, wie erwähnt, Steuersignale oder Daten sein, den FLASH Speicher. Dabei werden auch alle relevanten Speichertimings respektiert.

5.3.3.1 Ein- und Ausgänge Die Eingänge entsprechen den Ausgängen des Control Extractors, deren Funktionalität wurde bereits besprochen. Ebenfalls verfügt die FSM über ein Interface zu den beiden RS-232 Ports. Port 0 mit den Signalen txData1, txWr1 und txFull1 dient zur Kommunikation mit dem Host (Perl Script). Port 1 mit txData2, txWr2 und txFull2 kann als Debug Interface verwendet werden, z.B. um von gewissen Zuständen der FSM Testnachrichten an den Host zu sehen. Die Funktionalität der RS-232 Signale wurde ebenfalls bereits erklärt.

Der einzige neue Eingang ist das STS (Status) Signal. Dieses stammt von FLASH Chip und arbeitet in der Normalkonfiguration als Busy/Ready Signal. Der FLASH Chip verfügt intern ebenfalls über eine Zustandsmaschine. Diese interpretiert beispielsweise Befehle zum Block Erase und führt anschliessend den Löschvorgang durch. Die FLASH Zustandsmaschine aktualisiert das STS Signal um den aktuellen Status anzuzeigen. Das STS Signal kann folgende Zustände annehmen:

Zustand	Bedeutung
'0'	FLASH Busy, der Speicher kann keine neuen Befehle verarbeiten da er zur Zeit eine Operation ausführt.
'1'	FLASH Ready, der Speicher ist idle und steht für neue Aufgaben bereit.

Tabelle 16: FLASH Programmation - FSM - STS Eingang

Das STS Signal wird dementsprechend für Kontrollen der Ablaufsequenz der Zustandsmaschine eingesetzt. Zum Beispiel kann nach Abgabe eines neuen Befehls an den FLASH gewartet werden bis das STS Signal auf '0' wechselt. Das bedeutet, dass der Speicher den Befehl erhalten hat und ihn ausführt. Wechselt STS anschliessend zurück auf '1', ist die Operation abgeschlossen und es kann zum nächsten Zustand in der FSM übergegangen werden.

Die folgende Tabelle gibt einen Überblick über die restlichen Ausgänge dieses Blocks und deren Default Wert.

Signal/Bus	**Zweck**	**Default Wert**
data	16 Bit, bidirektionaler Datenbus zum und vom FLASH.	others => 'Z'
addr	24 Bit Adressbus für den FLASH. Auf dem Board werden addr(23 downto 1). verwendet.	others => '-'
flashReset	Reset Signal für den Speicher. Beim Start des Systems wird der Speicher zurückgesetzt und anschliessend bleibt er betriebsbereit. Wird auf dem Toplevel invertiert da es ein aktiv tief Signal ist.	'0'
ce	Chip Enable Signal für den Speicher (CE0). Aktiviert den Chip. Wird auf dem Toplevel invertiert da es ein aktiv tief Signal ist.	'0'
oe	Output Enable Signal für den Speicher. Zeigt einen Lesevorgang an. Wird auf dem Toplevel invertiert da es ein aktiv tief Signal ist.	'0'
we	Write Enable Signal für den Speicher. Startet einen Schreibvorgang. Wird auf dem Toplevel invertiert da es ein aktiv tief Signal ist.	'0'

Tabelle 18: FLASH Programmation - FSM - Kombinatorische Ausgänge

Dies sind zugleich beinahe alle Signale des FLASH Chips an sich. Er verfügt einzig über einen zusätzlichen BYTE Eingang und zwei weitere CE Signale. Diese sind aber alle fix verkabelt. Der BYTE Eingang kann zwischen einem 8 Bit und 16 Bit Modus wechseln. Ist aber fix auf 16 Bit verkabelt. Im 16-Bit Modus wird das Adressbit 0 nicht verwendet, dieses ist ebenfalls fix auf den Ground gezogen. Daher wird dieses, wie in der Tabelle zu sehen ist, im addr Signal auch nicht verwendet.

Folgende Abbildung zeigt den Block mit all seinen Ein- und Ausgängen.

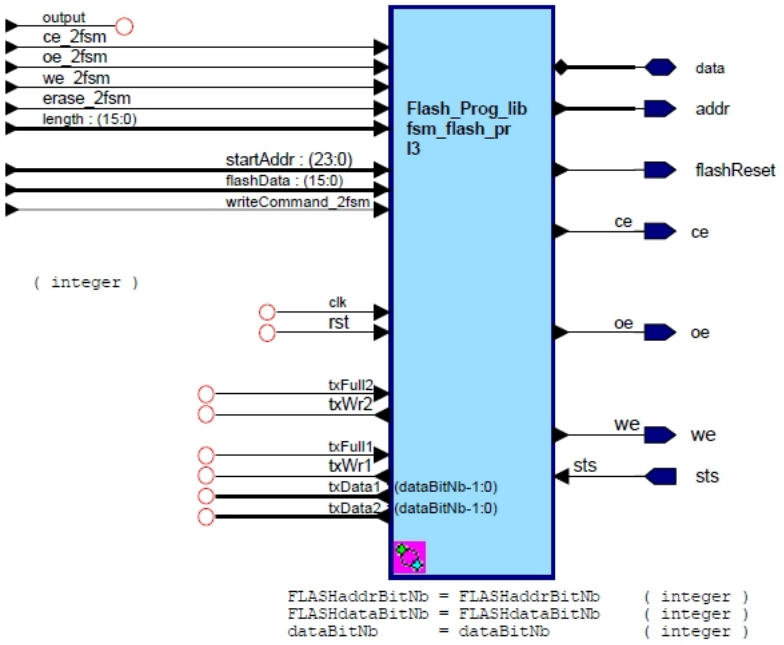


Abbildung 61: FLASH Programmation - FSM Block

5.3.3.2 Implementierung Die externen Signale wurden bereits in obenstehender Tabelle aufgezeigt. Der zugewiesene Default Wert ist wichtig, da für jeden Zustand der FSM alle kombinatorischen Ausgänge klar definiert sein müssen. Wird in einem Zustand ein Ausgang nicht mit einem neuen Wert versehen, wird automatisch der Default Wert übernommen. Dies erleichtert zudem die Entwicklung da ein kleinerer VHDL Code für jeden Zustand geschrieben werden muss.

Zusätzlich verfügt die FSM über folgende interne Signale, diese werden primär zur Zwischenspeicherung von bestimmten Werten eingesetzt.

Signal/Bus	Zweck	Reset Wert
eraseAddr	Enthält die Adresse des aktuell zu löschenden Blocks im Block Erase Modus.	others => 'Z'
lengthFSM	Enthält die aktuelle Länge der noch zu schreibenden bzw. lesenden 16 Bit Wörter. Wird nach jedem Schreib- oder Lesezyklus um -4 decrementiert.	others => '-'
maxEraseAddr	Enthält fix den Wert 0x1E0000, welches die Startadresse des letzten Blocks im 16 Bit Modus des FLASH ist.	'0'
txNibble1	Speichert ein Nibble des 16 Bit Datenwortes beim Lesen des Speichers, dies entspricht einem HEX Wert, welcher später in einem ASCII Charakter übersetzt wird.	'0'
writeAddr	Enthält die momentan zu lesende oder schreibende Adresse im Speicher. Wird nach jedem Zyklus um +2 inkrementiert.	'0'

Tabelle 20: FLASH Programmation - FSM - Interne Signale

Da diese Signale getaktete Signale sind, verfügen sie über einen Reset Wert. Dieser wird beim Start des Systems gesetzt. Zwischen den Zuständen behalten die Signale aber den aktuell zugewiesenen Wert.

Das Perl Script, wie später noch genau erklärt wird, gibt die zu übertragende Länge als Anzahl ASCII Charaktere an. Da der Control Extractor aus einem ASCII Wert einen HEX Wert macht, enthält ein 16 Bit Datenwort zum oder vom Speicher vier Charaktere. Dementsprechend wird die Länge jeweils um -4 decrementiert. Deshalb wird für die Rückwandlung auch das Signal txNibble1 verwendet, es kann einen vier Bit HEX Wert für die Rückwandlung in einen ASCII Wert sichern. Pro Lesezyklus werden also alle vier Nibbles des 16 Bit Datenwortes einzeln umgewandelt und per RS-232 an den Host geschickt. Erst danach wird zur nächsten Adresse gesprungen.

Die Adresse wird um +2 inkrementiert da es sich um einen 16 Bit Speicher handelt. Pro Adresse kann ein 16 Bit Datenwort zugegriffen werden. Folgende Abbildung zeigt dies auf.

...		
0x000004	2. Byte	1. Byte
0x000002	2. Byte	1. Byte
0x000000	2. Byte	1. Byte

Abbildung 62: FLASH Programmation - Adressierung 16 Bit FLASH Speicher

Darum wird im 16 Bit Modus das Adressbit 0 des FLASH Speichers auch nicht verwendet. Es können nur gerade Adressen vorkommen.

Folgende Abbildung zeigt grob den Ablauf der FSM.

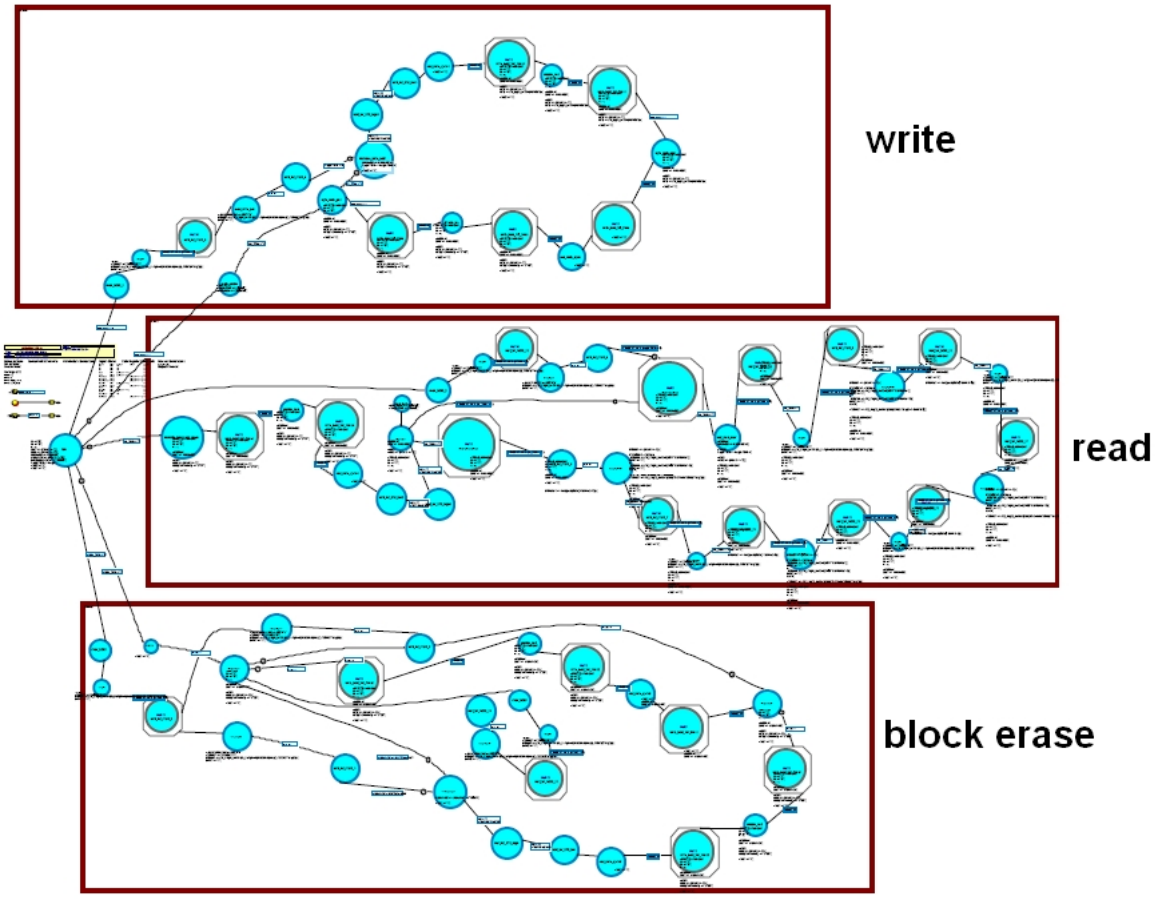


Abbildung 63: FLASH Programmation - FSM Ablauf

Die FSM ist zu Beginn im Zustand Idle (Zustand links, nicht eingerahmt). Von dort aus werden die verschiedenen Äste der FSM gestartet. Die Äste werden solange wiederholt, bis die Operation beendet ist. Folgende Tabelle gibt einen Überblick über die implementierten Äste.

Ast	Startsignal	Ausführung	Endbedingung
BLOCK ERASE	erase_2fsm = '1'	Löscht Block für Block des FLASH Speichers. Pro Zyklus müssen zwei Befehle an den Speicher übermittelt werden. Die Adresse muss sich innerhalb des zu löschenden Blocks befinden.	Aktuelle Löschadresse grösser als maximale Löschadresse.
WRITE	writeCommand_2fsm = '1'	Schreibt das aktuell erhaltene 16-Bit Datenwort in den Speicher. Dazu sind zwei Speicherzugriffe nötig, ein Schreibbefehl und die Übertragung der Daten. Die Adresse hat den Wert der Zieladresse im Speicher.	Länge der noch zu schreibenden Charaktere ist null.
READ	oe_2fsm = '1	Zu Beginn wird der Speicher wieder in den Read Array Modus versetzt. Anschliessend wird der Speicher gelesen, Nibble für Nibble des 16 Bit Datenwortes in einen ASCII Wert umgewandelt und per RS-232 an den Host verschickt.	Länge der noch zu lesenden Charaktere ist null.

Tabelle 21: FLASH Programmation - FSM Auslöser und Endbedingungen

Auf einen Ausdruck wird verzichtet, da die FSM zu gross ist. Eine genaue, detaillierte Betrachtung der FSM ist am besten mit der auf der CD beigelegten, digitalen Version möglich.

In jedem Ast wurden die Speichertimings gemäss Intel Datenblatt genau beachtet. Realisiert wurden diese Timingbedingungen mit Hilfe von Waitstates. Dies sind FSM Zustände, in denen die Anzahl der zu wartenden Clock-Perioden definierbar ist. Eine Clock-Periode dauert ca. 15ns. Im Intel Datenblatt sind Minimum- oder Maximumwerte vorgeschrieben, diese können durch Aufsummieren der Clock-Perioden erreicht werden. Z.B. muss für ein 50ns Timing mindestens vier Clock-Perioden gewartet werden.

$$t_{clock} = \frac{1}{66MHz} = 15.15ns$$

Beim Verlassen eines Waitstates kann die Timeout Bedingung zusammen mit einer anderen Bedingung (z.B. Wert eines Signals oder Registers) kombiniert werden. Die Timings werden im Testkapitel genauer betrachtet.

5.3.4 Bemerkungen zum Intel FLASH Speicher.

Zusammenfassend an dieser Stelle nochmals wichtige Informationen zum eingesetzten FLASH Speicher. Für genaue Informationen ist das Datenblatt[8] zu konsultieren.

- arbeitet fix im 16-Bit Modus, Adressbit 0 wird nicht verwendet
  - Adressierung mit +2er Adresssprüngen, es kommen nur gerade Adressen vor.
- verfügt über 32 64 Kword Blöcke von 0x000000-0x00FFFF, 0x010000-0x01FFFF bis 0x1F0000-0x1FFFFF.
- besitzt drei Chip Enable Eingänge, nur CE0 steht zur Verfügung
- STS Signal zur Hardwarekontrolle des Status der internen Zustandsmaschine
- besitzt zahlreiche interne Konfigurations- und Status-Register, welche in einem separaten Konfigurationsmodus adressiert werden können.
- Löschvorgang setzt alle Bits auf '1'
- siehe Test Kapitel für Bemerkungen zu den Timings

[8] Pfad P:\PCB\Produit\Logical\FPGA-EBS\FPGA_EBS_V2.0\Datasheets\flash - 28F320J3A.pdf oder CD zur Diplomarbeit

5.3.5 Perl Script für Host Computer

Das Perl Script wurde mit drei Teil-Scripts umgesetzt. So kann der Ablauf der Bedienung der Scripts sauber getrennt werden und es muss z.B. nicht für das Lesen des Speichers zuerst ein Löschvorgang ausgeführt werden. Die Namen der Scripts sind nummeriert, dies erleichtert erstens den korrekten Ablauf und zweitens das Starten der Scripts durch die Autovervollständigung. Gestartet werden die Scripts in einer Konsole (Cygwin, CMD oder andere Kommandozeilenkonsole) mit dem Befehl *perl <scriptname.pl>*.

Auf eine Codebeschreibung wird an dieser Stelle verzichtet. Die Scripts sind alle gut kommentiert. Es wird jedoch für jedes Script ein Flow Chart aufgezeigt und besondere, wichtige Punkte hervorgehoben.

5.3.5.1 Allgemeines Alle Scripts führen zu Beginn einige Includes durch. In Perl wird dies mit dem use Befehl gemacht. Bei den Includes handelt es sich um den Seriellen Port für die RS-232 Verbindung, ein Texteingabe Hilfsmittel sowie die Switch-Case Anweisung.

Danach folgen diverse Konstanten bzw. Defines wie z.B. Datenframelängen, ACK und NACK Werte. Daraufhin wird ein Objekt für den Seriellen Port erstellt. Mit diesem wird im Script die Verbindung aufgebaut. Nach der Deklaration der verwendeten Variablen, wird dieses RS-232 Objekt initialisiert. Dabei werden folgende Parameter gesetzt. Diese stimmen im Übrigen mit dem RS-232 Interface in der FPGA überein.

Parameter	**Wert**
Baudrate	9600
Parity	keine
Datenbits	8
Stopbits	1
Handshake	keiner

Tabelle 22: FLASH Programmation - Perl Script - RS-232 Einstellungen

Die Baudrate ist relativ langsam eingestellt. Bei diesem System sind keinerlei Sicherheitsmechanismen implementiert, z.B. Parity Checks. Um die Fehlerwahrscheinlichkeit tief zu halten, wurde eine langsamere Übertragung gewählt.

Nun gibt jedes Script noch eine Willkommensmeldung aus.

5.3.5.2 1_Erase_Flash.pl Folgendes Flussdiagramm zeigt den Ablauf des Scripts zum Löschen des Speichers auf.

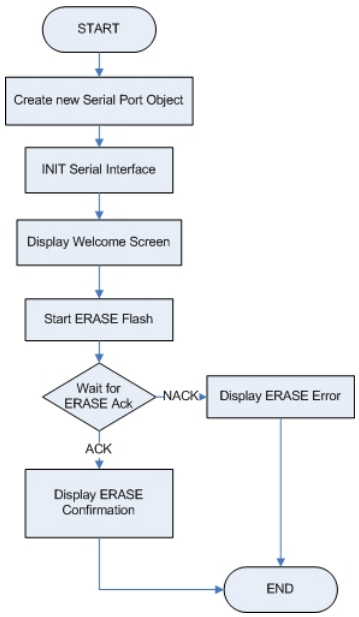


Abbildung 64: FLASH Programmation - Perl Erase Script - Flow Chart

Das Script muss einzig den Löschbefehl über das RS-232 Objekt senden, den Rest übernimmt die FSM in der FPGA. Während dieser Zeit wartet das Script auf eine Rückmeldung von der Zustandsmaschine. Sobald der Vorgang beendet ist, wird eine Bestätigung angezeigt. Beim Löschvorgang werden alle Bits im Speicher auf '1' gesetzt.

5.3.5.3 2_Write_Flash.pl Das Schreib-Script fragt den Benutzer, nachdem es die Willkommensmeldung angezeigt hat, nach der .srec Datei, welche geschrieben werden soll. Daraufhin wird die Datei in den Speicher, also in ein Array, gelesen. Das Array wird im Anschluss Linie für Linie, sprich S-Record für S-Record, durchlaufen. Dabei wird für jeden Record der Typ, die Adresse und dessen Datenwert erfasst. Da diese Werte, je nach Recordtyp, anders sein können, wird dieser Ablauf in einem Switch-Case, getrennt nach Recordtyp, realisiert. Die Daten werden in Blöcken von acht Zeichen geschickt. Bei grösseren Dateien kann die Übertragung über das langsame RS-232 Interface lange dauern.

Das Script geht also jeden Record durch und sendet dessen Daten genau an die Adresse, die im Record angegeben ist. Es werden allerdings nur die Daten verschickt, welche einen Adressbereich für den FLASH haben (siehe Memory Map), alle anderen Records werden ignoriert.

Folgende Abbildung zeigt das Flussdiagramm dieses Scripts auf.

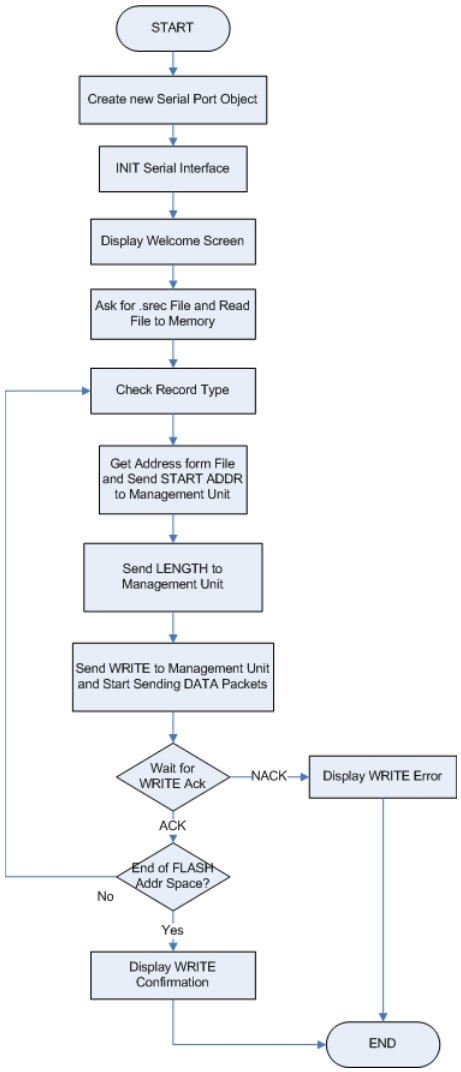


Abbildung 65: FLASH Programmation - Perl Write Script - Flow Chart

5.3.5.4 3_Read_Flash.pl Bei diesem Script wird der Benutzer nach dem Start nach einer Startadresse des Lesevorgangs und der Anzahl der S-Records, die gelesen werden sollen, gefragt. Diese Werte werden der FSM übermittelt. Danach startet der Lesevorgang durch Versenden eines Lesebefehls.

Dieses Script erstellt eine neue .srec Datei, in welcher die gelesenen Daten vom FLASH gespeichert werden. Diese Datei heisst *flashReadout.srec* und muss vorher im gleichen Verzeichnis wie die Scripts angelegt werden.

Achtung: Das Script überschreibt bei jedem Lesevorgang den gesamten Inhalt der Datei. Falls gewisse Daten beibehalten werden sollen, muss vor einem erneuten Start des Scripts eine Kopie der Datei erstellt werden.

Die Zieldatei wird nun geöffnet und es wird ein eigener S0 Record geschrieben. Daraufhin wird in einer Schlaufe, bis die Lesebestätigung eintrifft, ein S3 Record nach dem anderen in die Datei geschrieben. Dabei wird der Recordtyp und die Länge fix geschrieben. Da beim Lesen nicht bekannt ist, welcher Recordtyp sich im Speicher befindet, wird hier nur mit S3 Records gearbeitet. Die Adresse wird anhand der Startadresse errechnet und inkrementiert sich zwischen den Records jeweils um 16. Nun werden die empfangenen Daten Zeichen für Zeichen in die Datei geschrieben. Bis eine Länge von 32 Zeichen erreicht ist. Dann folgt ein Platzhalter für die Checksum und ein Zeilenumbruch.

Sobald die Lesebestätigung eintrifft, wird die soeben angefangene Zeile (Recordtyp, Länge und nächste Adresse) wieder gelöscht. Dazu muss die *flashReadout.srec* Datei aber erst wieder in ein Array eingelesen werden. Innerhalb des Arrays wird in die letzte Zeile gewechselt, welche daraufhin entfernt wird. Nun kann das modifizierte Array wieder in *flashReadout.srec* gespeichert werden. Am Ende wird eine Bestätigung angezeigt.

Untenstehend das Flussdiagramm des Scripts.

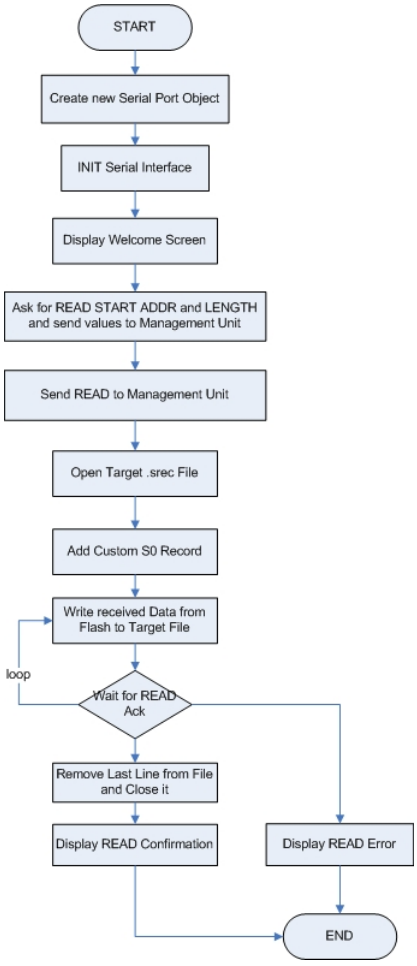


Abbildung 66: FLASH Programmation - Perl Read Script - Flow Chart

Alle drei Scripts sind in Beilage 19 angefügt.

5.3.5.5 Terminalbedienung Für kleine Tests kann auch direkt ein Terminal verwendet werden. Dazu muss die Terminalsoftware den COM1 Port verwenden. Diesem müssen die oben erwähnten RS-232 Parameter konfiguriert werden und die Verbindung kann aufgebaut werden. Ein lokales Echo ist zudem zu empfehlen. Die zusendenden Befehle sind im Spezifikationskapitel erwähnt.

Folgende Abbildung zeigt einen Löschvorgang ('x') inkl. empfangener Löschbestätigung ('o').

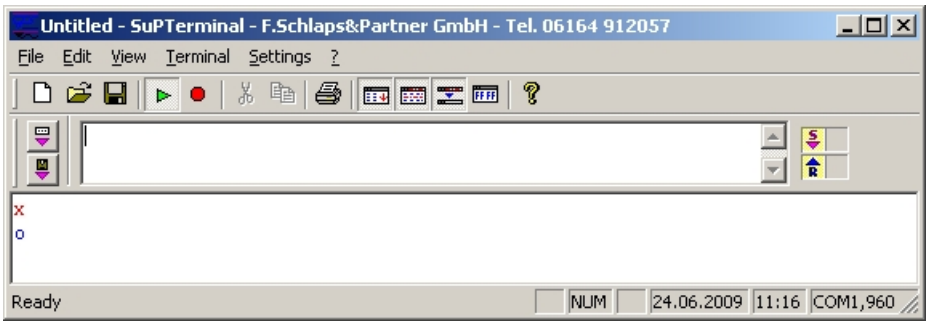


Abbildung 67: FLASH Programmation - Terminal Bedienung (1)

Ein kompletter Lösch-, Schreib- und Lesevorgang einer kleinen .srec Datei würde folgende Befehle und Daten verlangen:

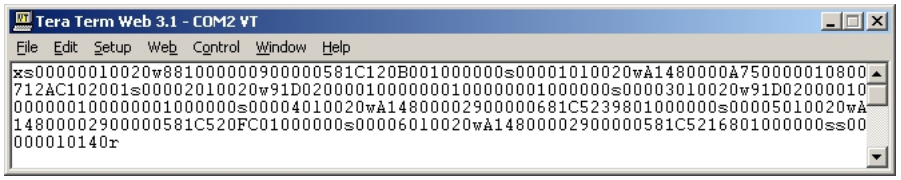


Abbildung 68: FLASH Programmation - Terminal Bedienung (2)

5.3.5.6 Installation von Perl Modulen In diesem Abschnitt wird erklärt wie ein Perl Modul nachinstalliert werden kann. Dies ist z.B. bei *Win32::SerialPort*, welches in den oben erwähnten Perl Scripts verwendet wird, nötig.

Nach einer normalen Installation von *Active Perl*, hier Version 5.10, kann in einer Konsole *CPAN* (Comprehensive Perl Archive Network) gestartet werden. Von dort aus lassen sich Module nachträglich downloaden und installieren. Dazu ist der Befehl *install <libName>::<moduleName>* einzugeben, für das serielle Interface konkret:

install Win32::SerialPort. CPAN lädt anschliessend die Source Dateien herunter und installiert diese.

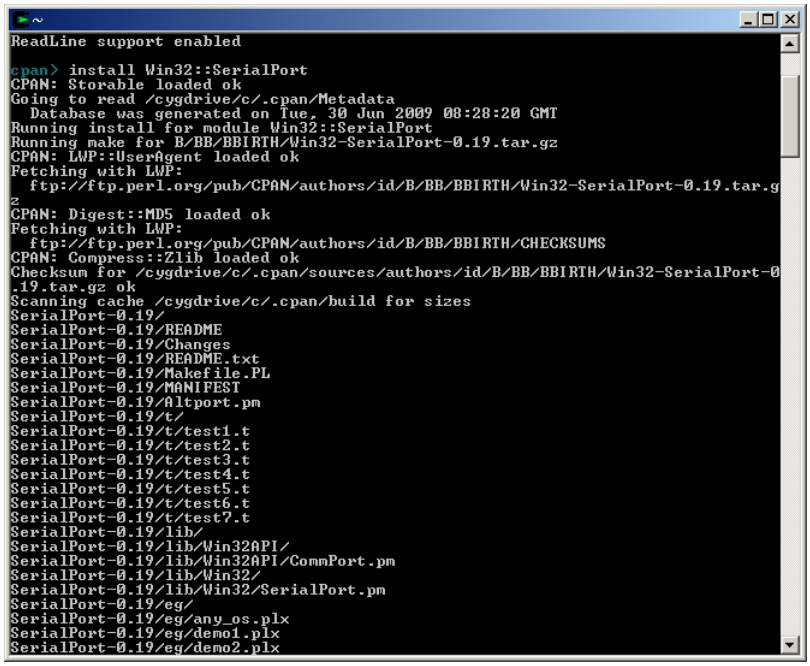


Abbildung 69: CPAN Perl Modul Installation

Es kann vorkommen, dass das Modul nicht korrekt installiert werden kann. In einem solchen Fall können die Sourcedateien (.pm) manuell vom CPAN Verzeichnis ins Perl Library Verzeichnis kopiert werden. Z.B. für das Serielle Interface von

C:\.cpan\build\SerialPort-0.19\lib inkl. Unterverzeichnisse *Win32\SerialPort.pm* und *Win32API\CommPort.pm*

nach

C:\Perl\lib

Anschliessend kann das Modul nun in den Perl Scripts verwendet werden.

5.3.6 Simulation

Anschliessend kann die FLASH Schaltung simuliert werden. Die Simulation ist ein wichtiger Schritt um die Funktionalität des Systems, vor dessen Implementation auf der Hardware, zu testen und allenfalls zu korrigieren.

Alle durchgeführten Simulationen sind im Kapitel Tests beschrieben.

5.3.7 Synthese und Place & Route

Für die Synthese, das Mapping und das Place & Route kann für diese Schaltung problemlos Xilinx ISE eingesetzt werden.

5.3.7.1 Prepare for Synthesis Im HDL Designer wird im Design Manager die Library *Flash_Prog_Toplevel* geöffnet und die Schaltung *toplevel* ausgewählt. Nun kann rechts *Prepare for Synthesis* gestartet werden. Zuvor sind aber, wie im LEON Kapitel beschrieben, die User Variables anzupassen.

5.3.7.2 Xilinx ISE Nun kann Xilinx Project Navigator doppelgeklickt werden und es öffnet sich Xilinx ISE und lädt automatisch dessen Projekt-File. Beim ersten Start, oder nach manuellem Erstellen eines neuen Projekts, muss die verwendete FPGA definiert werden. Ein Doppelklick auf das Chip-Icon öffnet folgendes Fenster:

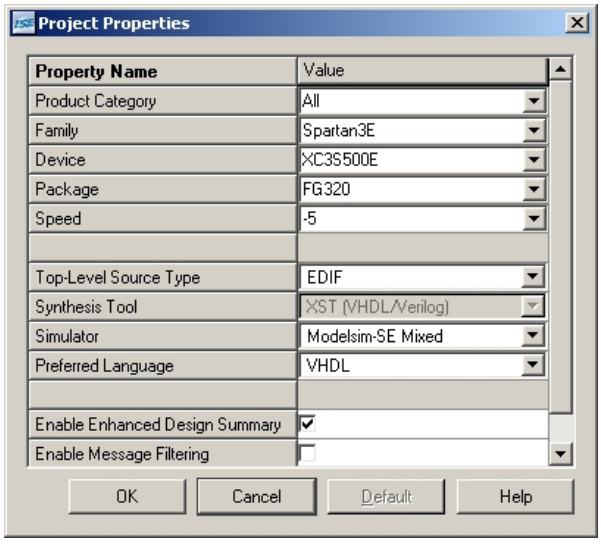


Abbildung 70: Xilinx ISE - FPGA Einstellungen

Es muss die Produkt Familie, das Gerät, das Paket und die Geschwindigkeit, wie in der Abbildung zu sehen, eingestellt werden. Ebenfalls muss als bevorzugte Sprache VHDL gewählt werden.

ISE braucht für dieses Projekt zwei Dateien, die .vhd Datei mit der Schaltung und eine .ucf Datei. Die .ucf Datei wird, falls sie nicht vorhanden ist, von ISE erstellt und dem Projekt hinzugefügt. In der .ucf Datei sind wichtige System- bzw. Schaltungsinformationen gespeichert, z.B. die Pinbelegung oder Timing Zwänge (Constraints). Ein solches Constraint kann beispielsweise die maximale Länge eines Signalroutes innerhalb der FPGA sein, welche durch die Clockperiode limitiert wird. Eine korrekte .ucf Datei kann gesichert und wiederverwendet werden. Der Vorteil ist, dass z.B. die Pinbelegung so nur ein Mal gemacht werden muss.

Dem Projekt wird nun die .vhd Datei durch *Rechtsklick* ⇒ *Add Source...* aus dem */concat* Verzeichnis hinzugefügt. Daraufhin sollte ISE folgende Dateien anzeigen:

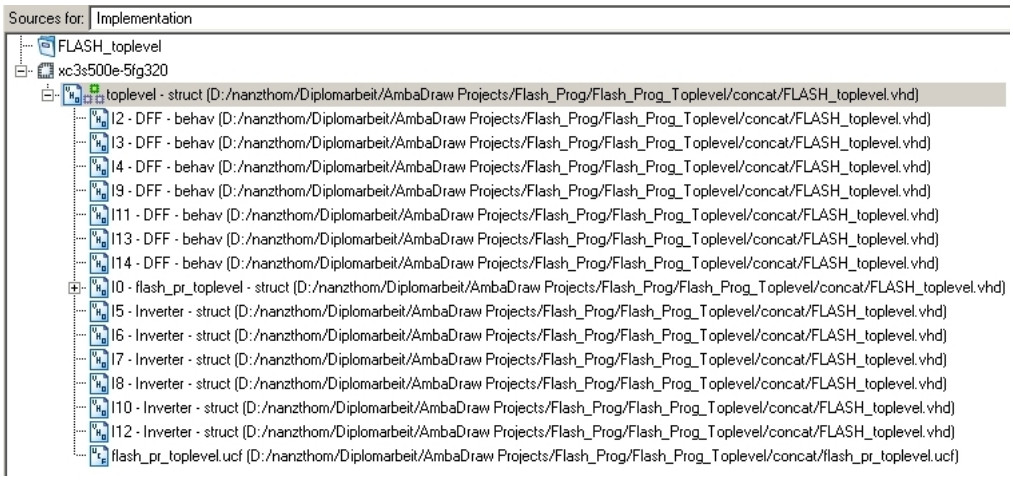


Abbildung 71: Xilinx ISE - Sources

Die Hauptschaltung (*FLASH_ toplevel.vhd*) sowie das .ucf (*flash_ pr_ toplevel.ucf*) sind zu sehen. Die restlichen Elemente sind die Flip Flops und Inverter des Toplevels.

Im unteren Teil des ISE Fensters sind die Operationen und Befehle zu sehen. Wählt man bei den Sources die Hauptschaltung aus, sind folgende Befehle ausführbar:

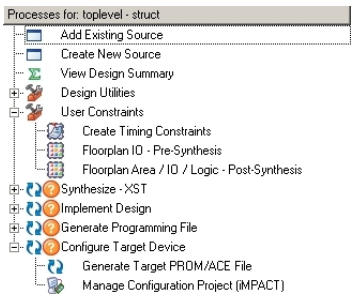


Abbildung 72: Xilinx ISE - Processes

Im Gegensatz zur LEON Schaltung ist hier der Befehl *Synthesize - XST* auch verfügbar und wird auch verwendet.

Unter *User Constraints* ⇒ *Floorplan Area / IO / Logic - Post-Synthesis...* kann nun die Pinbelegung der FPGA angepasst werden. Dies erfolgt gleich wie bei der LEON Schaltung mit dem Unterschied, dass hier kein synplicity.ucf verwendet wird und dieses auch nicht bei jeder neuen Revision zu modifizieren ist. In diesem ISE Projekt wird immer das gleiche .ucf verwendet da die Constraints nicht variieren. Es wird empfohlen das fertige .ucf zu sichern, wie in diesem Fall im */concat* Verzeichnis. Das .ucf ist in Beilage 20 angefügt.

Folgende wichtige Punkte sind zu beachten:

- Signal rstn mit Pullup Abschluss (Button A15)
- Signal sts mit Pullup Abschluss gemäss Intel Datenblatt
- Adressen 1:1 von FPGA auf FLASH gemappt, A0 auf einen anderen Pin der FPGA gelegt

5.3.8 Download auf FPGA

Nun kann die FPGA mit der neuen Schaltung programmiert werden. Dies erfolgt gleich wie bei der LEON Schaltung. Allerdings ist hier nicht geplant das EEPROM zu schrieben, da die FLASH Schaltung nur eine temporäre Schaltung ist. Es wäre dennoch möglich, falls dies gewünscht ist.

Die Schaltung ist jetzt einsatzbereit in der FPGA und der FLASH kann gelöscht, gelesen oder geschrieben werden.

6 Tests

Dieses Kapitel fasst alle durchgeführten Tests inkl. deren Resultate zusammen. Alle Aufgaben der Diplomarbeit werden hier besprochen, angefangen mit dem LEON System.

6.1 LEON System

Die Tests zum LEON System befassen sich nur mit der Speicheransteuerung. Dies war bis zum Ende der Diplomarbeit die Hauptschwierigkeit, da der 16 Bit Zugriff auf den SDRAM nicht korrekt funktioniert. Nun folgend werden Simulationen und anschliessende Funktionskontrollen auf der Hardware besprochen.

Für beide Tests wurde eine kleine Applikation in C geschrieben. Dazu zuerst ein paar Bemerkungen.

6.1.1 Testapplikation, Compile & Link

Um die Speicheransteuerung zu testen, wurde in C ein kleines Testprogramm readtest.c geschrieben. Dieses ist in Beilage 21 angefügt. Wie zu sehen ist, werden zu Beginn zwei Defines erstellt. Diese enthalten die Speicheradressen für den RAM Bereich sowie die Konfigurationsregister des Speichercontrollers MCTRL. In der main-Schlaufe werden zuerst Pointer deklariert auf die oben definierten Adressen initialisiert. Im Anschluss werden alle drei MCTRL Konfigurationsregister geschrieben. Dies erfolgt so, wie es bereits im Realisationskapitel beschrieben wurde. Zu Testzwecken wird nun bei Adresse 10 der Wert 50 geschrieben. Diese Adresse wird im Anschluss ausgelesen und der Wert gespeichert. Mit Hilfe einer if-Schlaufe wird kontrolliert, ob der gelesene Wert mit dem geschriebenen Wert übereinstimmt. Entsprechend werden weitere, unterschiedliche Schreib- und Lesevorgänge durchgeführt. Am Ende bleibt das Programm in einer Endlosschleife hängen.
Damit dieses Programm in der Simulation sowie auch auf der Hardware verwendet werden kann, muss es kompiliert, gelinkt und in ein .srec Format umgewandelt werden. Dazu sind folgende Befehle in einer Konsole (z.B. Cygwin) auszuführen. Vorher muss in der Konsole allerdings ins Verzeichnis der Sourcedatei gewechselt werden.

Algorithm 2 Compile & Link

```
sparc-elf-gcc -msoft-float -c -g -O2 <filename>.c
sparc-elf-mkprom -freq 66 -rmw <filename>.o -msoft-float
sparc-elf-objcopy -O srec prom.out <filename>.srec
```

Die nun generierte .srec Datei, welche primär S3 Records enthält, kann in das *work* Verzeichnis der Testbench kopiert werden. Ebenfalls ist es möglich die Datei per Perl Script der FLASH Schaltung in den FLASH zu laden.

6.1.2 Simulation

Für die Simulation wurde eine neue Library erstellt, *testbench*. Diese enthält ein Blockdiagramm mit folgender Schaltung:

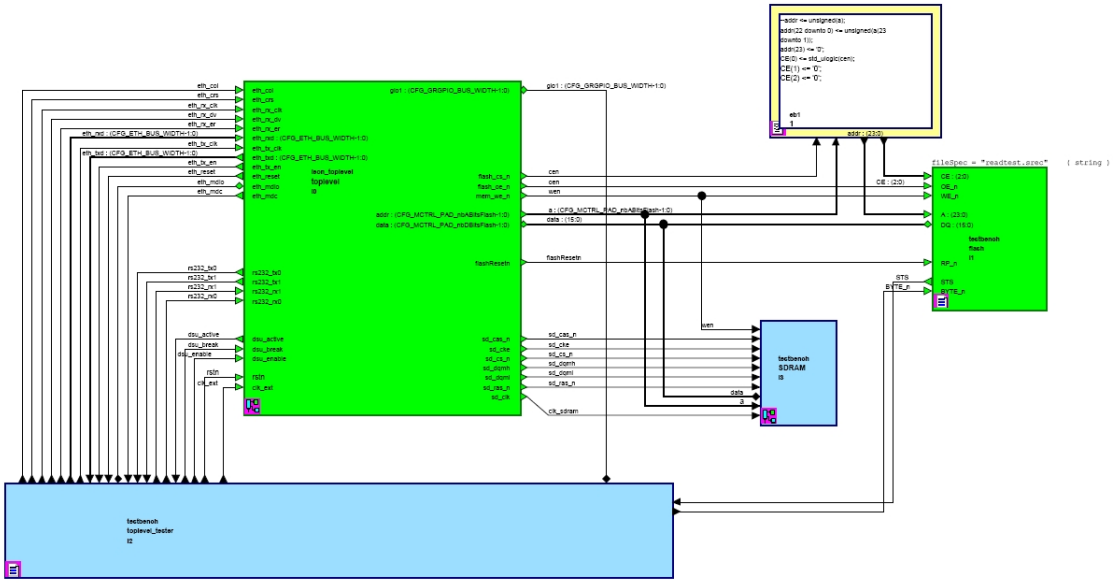


Abbildung 73: LEON Testbench

Der mittig platzierte grüne Block ist das LEON Toplevel inkl. allen Pads, Invertern und Synchronisierungen. Unterhalb im blauen Block ist das Testprogramm untergebracht. Daran werden alle nicht anderweitig verwendeten I/Os angeschlossen. Das Testprogramm generiert im Prinzip nur den Clock und den Reset der Schaltung. Ebenfalls werden die DSU Debug Signale gesetzt und das Byte Signal des FLASH Speichers zur Auswahl des 16 Bit Modus. Aufgrund des simplen Umfangs wird auf einen Ausdruck verzichtet.

Der FLASH Speicher wird mit dem von der HES-SO Wallis zur Verfügung gestellten Modell simuliert. Weitere Informationen dazu im Testkapitel zur FLASH Schaltung. Dem Block wird das zuvor generierte .srec File per Generic gesetzt. Im Embedded Block oberhalb werden dem FLASH zusätzlich die nicht verwendeten CE Bits gesetzt und die Adressen gemappt. Für die Simulation muss zwingend ein Adressoffset von 1 (*addr (22 downto 0) <= a(23 downto 1);*) gewählt werden. Ansonsten werden nur alle geraden Adressen im .srec ausgelesen und die geladenen Daten sind für den LEON nicht interpretierbar.

Im kleinen blauen Block befindet sich in einem Unterschaltbild der SDRAM:

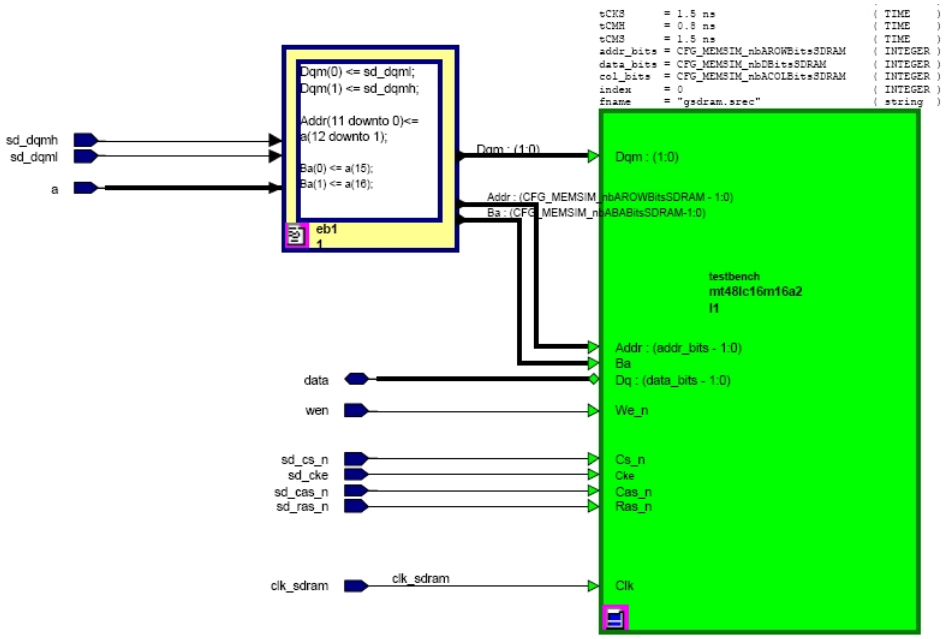


Abbildung 74: LEON Testbench SDRAM

Der Speicher wird mittels eines Micron Modells simuliert. Dieses Modell wird in der GRLIB mitgeliefert und entspricht bis auf die Anzahl der verfügbaren Adressen dem verbauten Micron Chip. Diesem werden nun alle Steuersignale wie CS, CAS, RAS usw. zugeteilt. Das Adressbit 0 wird, genau wie im Schaltplan für das Board, nicht verwendet. Daher die Zuweisung *Addr(11 downto 0) <= a(12 downto 1);*. Die DQM Signale zur Bytewahl innerhalb des Datenbusses (Datenmaskierung) sowie die Bankadressbits werden ebenfalls im Embedded Block gemappt. Die Bankadressbits sind allerdings auf einen Wert eingestellt, der nicht verwendet wird. Daher wird bisweilen nur die Bank 0 angesteuert. Der Block braucht ebenfalls eine .srec Datei im *work* Verzeichnis. Diese wird allerdings während der Simulation nicht verwendet, das Generic muss trotzdem gesetzt werden. Dem Modell des Speichers können auch zahlreiche Timings gesetzt werden.

Nun werden die durchgeführten Simulationen und deren Resultate analysiert.

6.1.2.1 Systemstart und Lesen des FLASH Speichers Der erste Test zeigt den Start des LEON Systems auf. Dies ist in folgender Abbildung zu sehen.

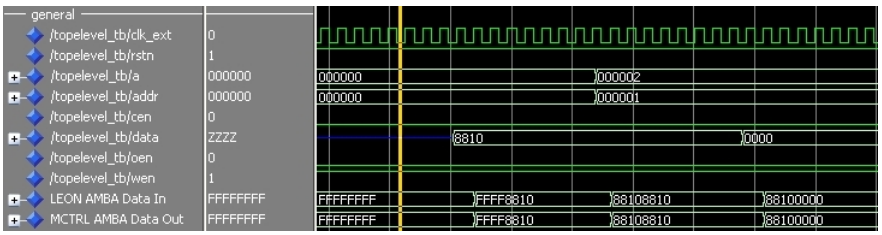


Abbildung 75: LEON Start und Lesen des FLASHs

Die erste gelesene Adresse entspricht 0x0, das ist auch der konfigurierte Resetvektor des LEON Prozessors. Im 16 Bit Modus adressiert der Speichercontroller in 2er Sprüngen. Durch den Offset in der Testbench werden alle Adressen halbiert und daher kann in 1er Sprüngen jede Adresse im .srec File angesprochen werden. Die ersten beiden 16 Bit Werte sind 0x8810 und 0x0000. Diese Werte werden nun von MCTRL über den AMBA Bus an den LEON übertragen, dies ist auf den untersten Signalen zu sehen.
Das System arbeitet nun die .srec Datei ab. Dabei wird Adresse für Adresse gewählt und der entsprechende Wert eingelesen. Die Werte können Befehle aber auch Daten sein. Bei einem Branch Befehl wird z.B. ein Adresssprung ausgeführt:

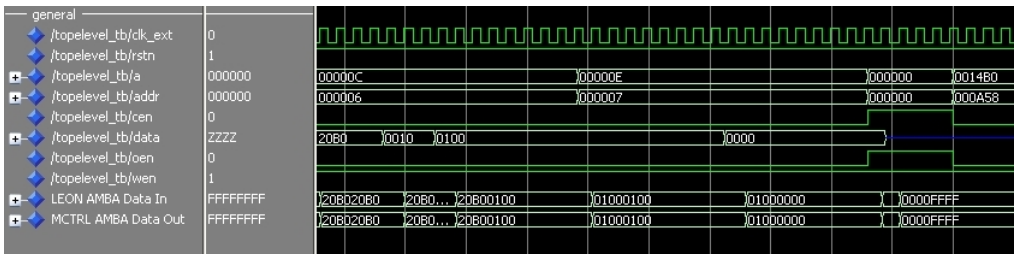


Abbildung 76: Adresssprung

In der obenstehenden Abbildung ist nochmals der Verlauf von Adressen und Daten sowie deren Übertragung auf dem AMBA Bus zu sehen. Am Ende ist ebenfalls ein zuvor angesprochener Adresssprung dargestellt.
Wie in der .c Testdatei definiert, werden vor dem Programmstart einige Register konfiguriert. Zusätzlich wurden vom Compiler/Linker weitere Konfigurationsbefehle hinzugefügt. Das eigentliche Schreiben und Lesen des SDRAMs beginnt erst nach einer Weile, die Verzögerung ist in folgender Simulation zu sehen.

Abbildung 77: Programmstartverzögerung

Wie oben dargestellt, wird der SDRAM nach der Systemkonfiguration aktiv. Die periodischen Aktivierungen des SDRAMs sind die REFRESH Befehle. Somit zeigen diese Tests auf, dass die .srec Datei korrekt aus dem FLASH gelesen wird, alle Register wie gewünscht gesetzt werden und anschliessend der Zugriff auf den SDRAM erfolgt. Dazu nun weitere Tests.

6.1.2.2 SDRAM Zugriff Folgende Abbildung gibt einen besseren Überblick über die Zugriffssequenz auf dem SDRAM.

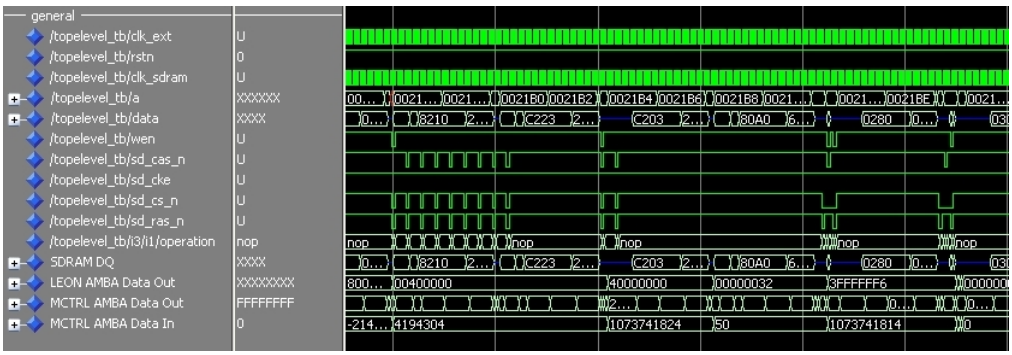


Abbildung 78: SDRAM Zugriffssequenz

Zu Beginn erfolgt die Initialisation des Speichers, dazu wird ein PRECHARGE Befehl gefolgt von einer Serie von AUTO-REFRESH Befehlen an den Speicher geschickt. In der Mitte ist ein LOAD MODE REGISTER Befehl zu sehen.

Interessanter sind allerdings die Schreib- und Lesezugriffe weiter rechts. Untenstehend ist der Schreibzugriff vergrössert dargestellt.

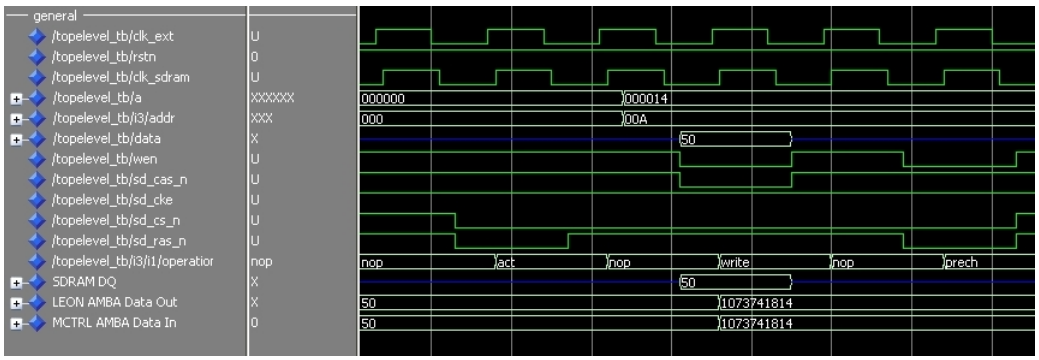


Abbildung 79: SDRAM Schreibzugriff

Der Schreibzugriff besteht, wie im Realisationskapitel gesehen, aus drei Etappen: einem ACTIVE, einem WRITE und einem abschliessenden PRECHARGE Befehl. Die Werte der Steuersignale CS, CAS, RAS und WE bestimmen den aktuellen Befehl. Während dem WRITE Befehl ist auf dem Adressbus des SDRAMs dank dem LSB Offset von MCTRL und dem Adressbusmapping in der Testbench die Adresse 10 (0xA) angelegt. Auf dem Datenbus befindet sich der Wert 50. Dieser ist so im .c File definiert und wird vom LEON via MCTRL über den AMBA Bus bereit gestellt. Zuletzt befindet sich der Datenwert auf dem SDRAM DQ Signal und wird im SDRAM gespeichert.

Anschliessend wird die gleiche Adresse wieder gelesen, dazu folgende Abbildung:

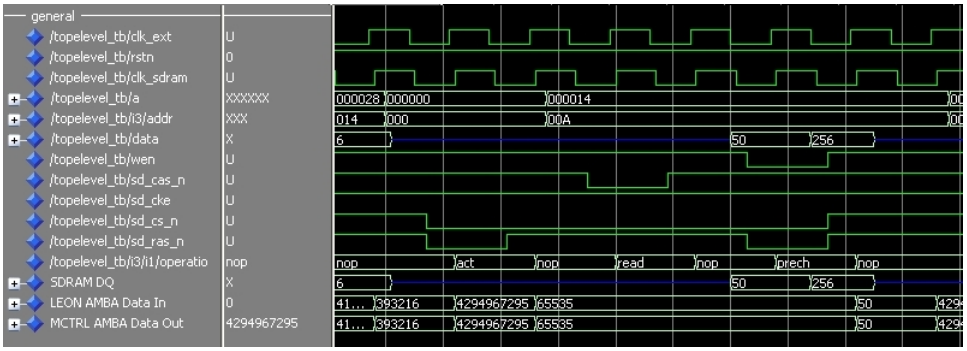


Abbildung 80: SDRAM Lesezugriff

Auch ein Lesen des SDRAMs beginnt mit einem ACTIVE, gefolgt von einem READ und einem PRECHARGE Befehl. Die Zeit zwischen ACTIVE und READ wird durch das tRCD Timing bestimmt. Es vergehen, wie konfiguriert, zwei SDRAM Clock Perioden bevor der READ Befehl angelegt wird. Gleiches gilt für das CAS Timing, d.h. die Verzögerung bis die Daten, Wert 50, auf dem Bus vorhanden sind. Daraufhin werden die Daten an MCTRL und letztendlich an den LEON übermittelt.

Somit kann aufgezeigt werden, dass Schreib- und Lesezugriffe auf den SDRAM funktionieren. Weitere Tests haben gezeigt, dass bei Zugriffen auf benachbarte Adressen keine Burst- sondern Einzelzugriffe durchgeführt werden. Diese Tests sind hier allerdings nicht dargestellt.

Gemäss .c Testprogramm erfolgt nun der Test ob der geschriebene und der gelesene Wert identisch sind. Beide obenstehenden Abbildungen zeigen dies auf. Jedoch führt der LEON Prozessor dennoch die else Bedingung aus. Leider reichte die Zeit nicht mehr aus um eine Lösung für dieses Problem zu finden. Der Schreibbefehl in der else Bedingung zeigt aber einen weiteren interessanten Punkt auf. Bei diesem Befehl wird ein 32 Bit Wert geschrieben. Bei dem verwendeten 16 Bit Speicher sollte MCTRL daher zwei 16 Bit Zugriffe durchführen.

Wie folgende Abbildung zeigt, wird jedoch nur ein Zugriff ausgeführt:

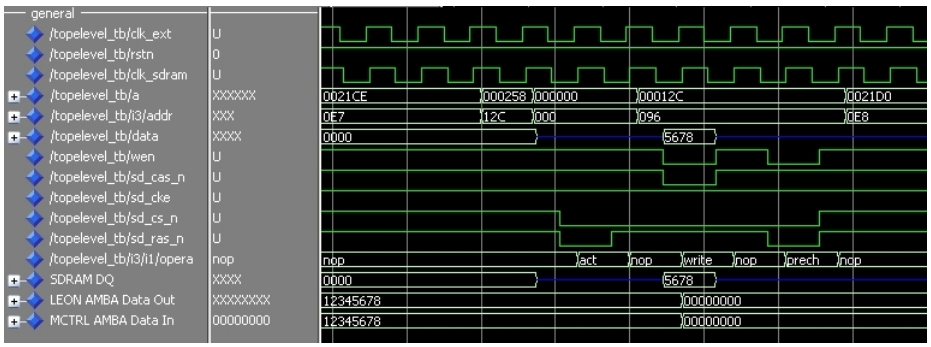


Abbildung 81: SDRAM 32 Bit Schreibzugriff

Nur die letzten 16 Bit werden geschrieben. Obwohl vom LEON Prozessor der korrekte 32 Bit Wert an MCTRL übertragen wird, führt MCTRL nur einen Schreibzugriff aus. Wobei das sdo.dqm Signal von MCTRL für zwei Zugriffe konfiguriert worden wäre. Dieses Problem konnte bis noch nicht gelöst werden.

Nach der Simulation wird die Schaltung in der FPGA getestet.

6.1.3 GRMON & Hardwaretests

Um die Schaltung auf der Hardware zu testen wird der Debugmonitor von Gaisler, GRMON, verwendet. Dieser liegt in einer 21 Tage Testversion vor und verfügt über zahlreiche sehr nützliche Hilfsmittel zum Debuggen eines LEON Systems und der darauf laufenden Applikation. In Cygwin kann GRMON mit *grmon-eval.exe* gestartet werden. Genaue Informationen zur Bedienung und zu den Befehlen sind dem GRMON Handbuch[9] zu entnehmen.

GRMON wird in diesem Fall über RS-232 betrieben, d.h. der AHBCTRL Port (J9) wird an COM1 des Hostcomputers angeschlossen. Vor dem Start von GRMON sollte allerdings das LEON System geresetet werden.

Beim Start von GRMON scannt die Software das angeschlossene LEON System und gibt die gefundene Konfiguration aus. Mit dem Befehl *info sys* kann eine erweitere Ansicht der gefundenen Komponenten aufgerufen werden.

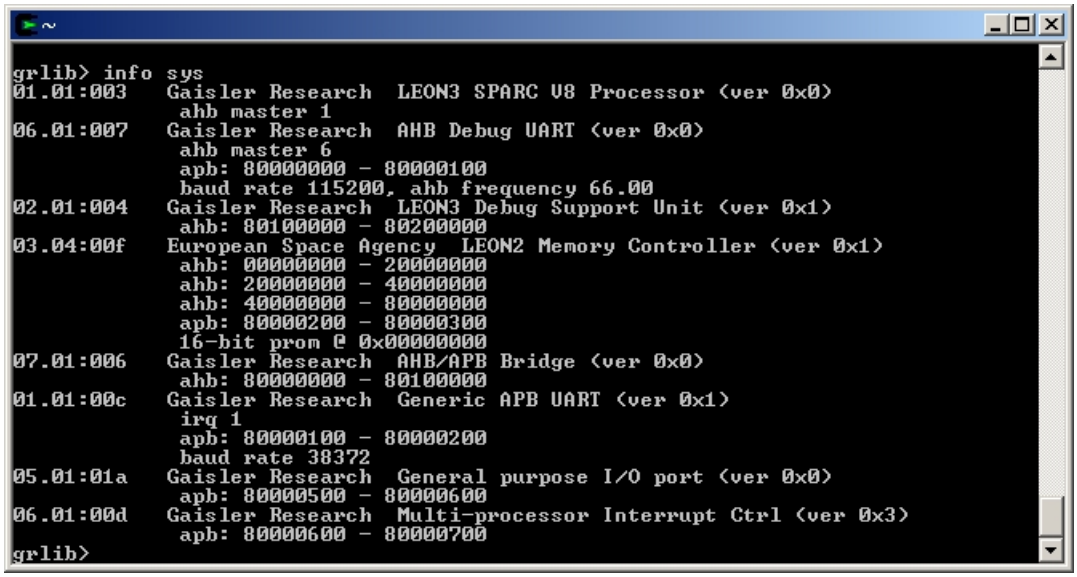


Abbildung 82: GRMON *info sys*

Die Komponenten entsprochen alle den aktiven IP Cores. Ebenfalls wurden alle Konfigurationsregister, wie in der Memory Map geplant, erkannt. Somit sind auch die Generics für die AHB und APB Adressierung und Maskierung korrekt gesetzt.

[9] grmon.pdf auf CD

Als erstes wird der FLASH gelesen, dazu ist der Befehl *mem 0x0 100* nötig. Der Parameter 100 bezieht sich auf die zu lesende Länge.

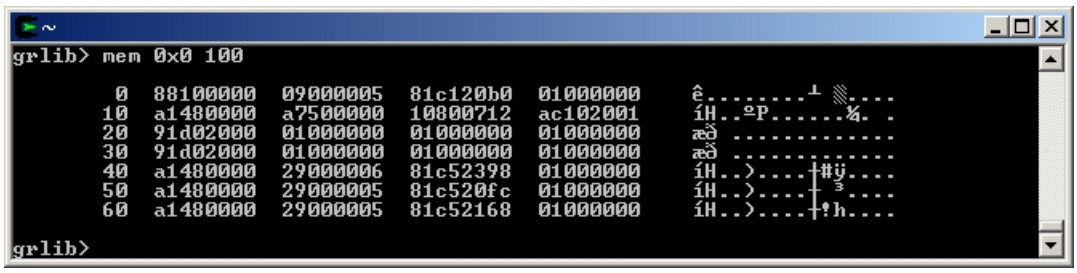


Abbildung 83: GRMON Lesen des FLASH Inhalts

Der Inhalt entspricht der geschriebenen .srec Datei. Der Code lässt sich auch deassemblieren:

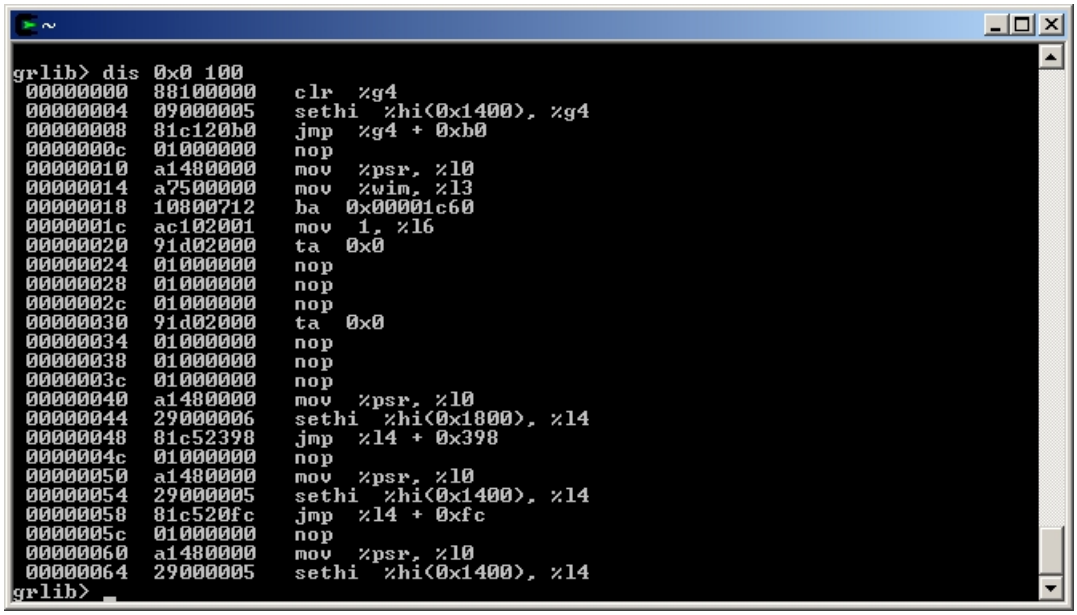


Abbildung 84: GRMON Deassemblieren des FLASH Inhalts

Wie zu sehen ist, kann GRMON die Befehle interpretieren, somit sind die Daten im FLASH gültig und korrekt.

Mit *info reg* können die Inhalte aller Konfigurationsregister ausgelesen werden. Dabei wurde festgestellt, dass die in der .srec Datei geschriebenen Werte für die MCTRL Register nicht übernommen wurden. Noch ist unklar ob die .srec Datei falsch gelesen wurde, auf Grund des fehlerhaften 32 Bit Schreibens das Register falsch gesetzt wurde oder ob GRMON beim Start einen Defaultwert setzt. Jedoch lassen sich die Register auch innerhalb von GRMON schreiben. In untenstehender Abbildung sind die dazu nötigen Befehle zu sehen.

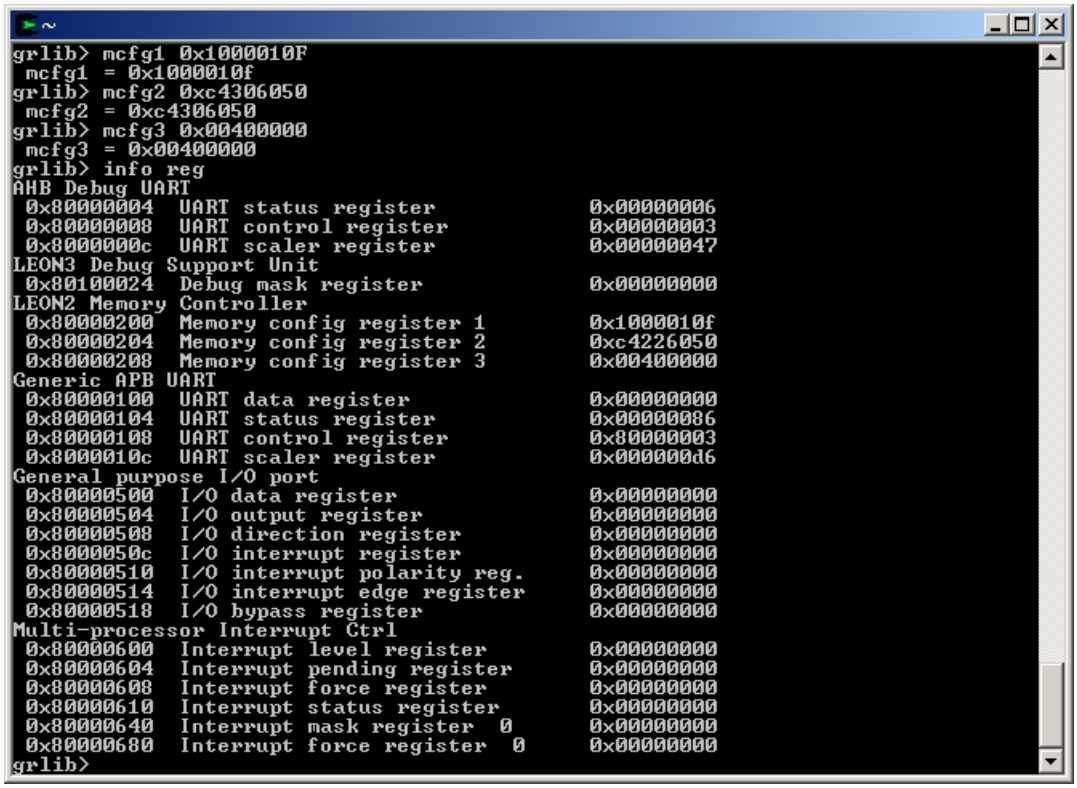


Abbildung 85: GRMON MCFG Register

Die Abweichung einiger Stellen des MCFG2 Registers sind auf Read-Only Bits zurückzuführen.

Nun kann der SDRAM gelesen und geschrieben werden. Dies zeigt folgende Abbildung auf.

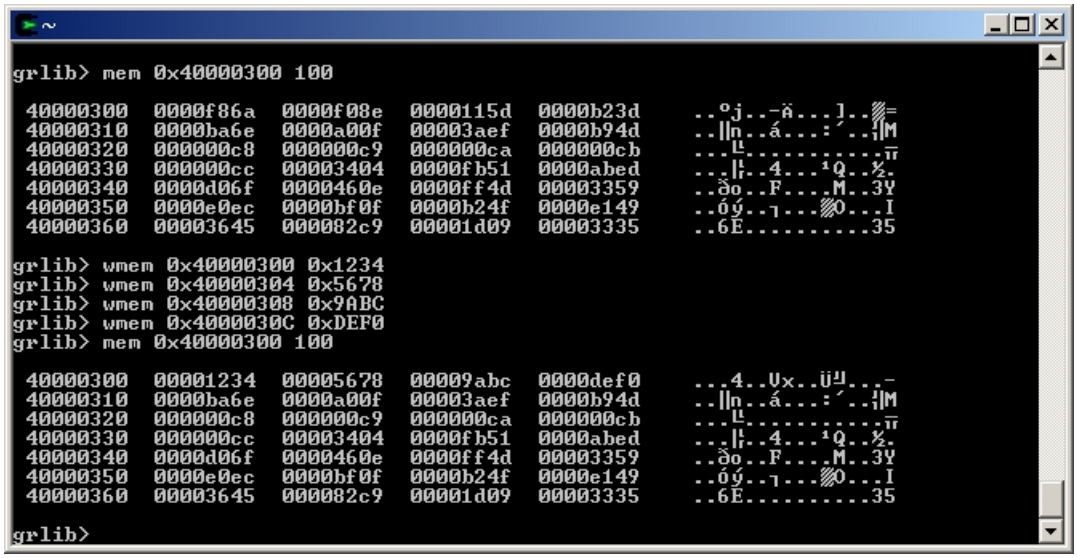


Abbildung 86: GRMON SDRAM Read/Write

Es wird zuerst mit dem *mem* Befehl eine beliebige Speicherposition gelesen. Der gefundene Inhalt ist zufällig. Anschliessend werden Daten an vier Adressen geschrieben, dazu kann der *wmem* Befehl verwendet werden. Der Adressbereich wird daraufhin nochmals gelesen. Die geschriebenen Daten sind im Speicher übernommen worden. Jedoch besteht auch hier das 32 Bit Problem. Es kann kein 32 Bit Schreibbefehl korrekt ausgeführt werden.

6.1.4 Schlussfolgerung

In der Simulation sowie auch auf der Hardware sind nur 16 Bit Zugriffe korrekt ausführbar. MCTRL übersetzt momentan 32 Bit Zugriffe nicht in zwei aneinander folgende 16 Bit Speicheransteuerungen. Ebenfalls werden die Konfigurationsregister nur in der Simulation wie gewünscht übernommen. Bei den GRMON Tests liegt ein Fehler vor.

Das restliche LEON System, z.B. der GPIO IP Core wurde nicht getestet. Dazu müssen zuerst die Speicherzugriffe korrigiert werden.

6.1.5 Verbesserungsmöglichkeiten

Folgende Punkte sind bei der LEON Schaltung für zukünftige Versionen zu verbessern:

- 32 Bit Schreib- und Lesezugriffe ermöglichen. Dies kann über eine korrigierte MCTRL Konfiguration und/oder über andere Adress- und Datenbus Mappings (Signal Routes Block) erreicht werden. Es fehlte am Ende der Diplomarbeit leider die Zeit diese Arbeit zu beenden.
- GRMON Registerkonfigurationsfehler beheben. Grund für den falschen Registerwert finden und allenfalls GRMON umkonfigurieren, z.B. Defaultwerte deaktivieren oder per Start-Batchdatei die Register automatisch schreiben lassen.
- FPGA Auslastung optimieren, wie bereits erwähnt durch Einsatz von Block RAM beim GRETH Core.

6.2 FLASH Programmation

6.2.1 Simulation der FPGA Schaltung

Um die korrekte Funktionalität der FLASH Programmationsschaltung sicher stellen zu können, wurden umfangreiche Simulationen durchgeführt. Dank den Simulationen war es möglich Fehler im Ablauf und in der Ansteuerung des Speichers frühzeitig zu erkennen und zu korrigieren. Sobald die Simulation ein korrektes Resultat lieferte, konnte mit der Synthese, wie sie im Kapitel zur Realisation beschrieben ist, begonnen werden.

Folgende Testbench wurde für die Simulation verwendet.

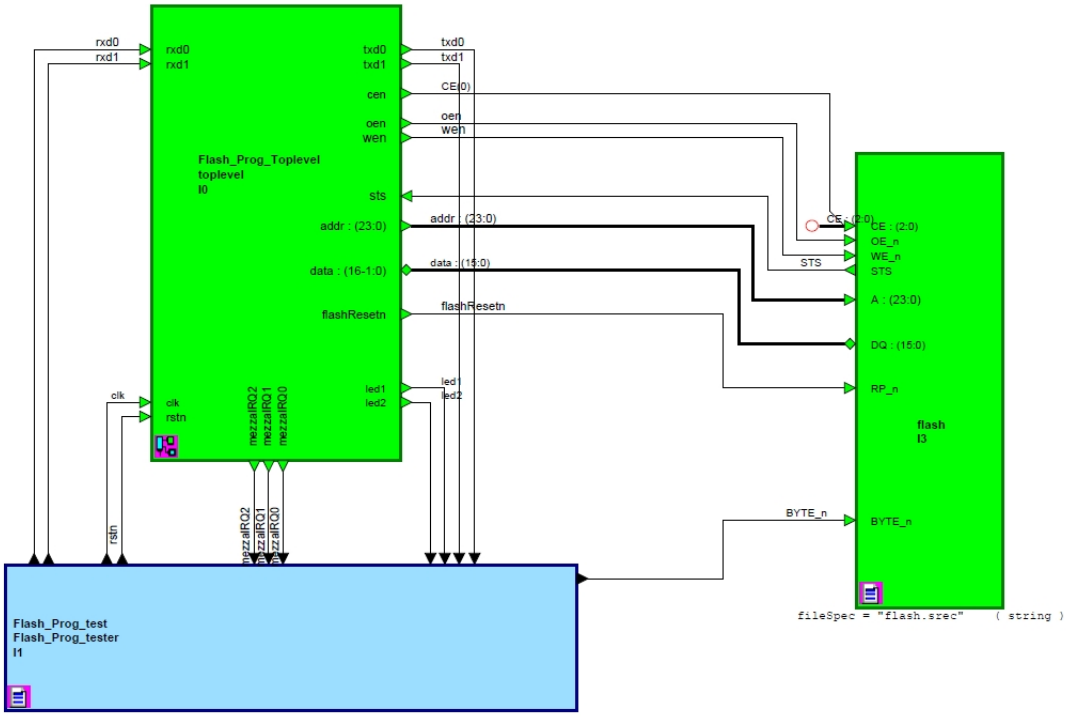


Abbildung 87: FLASH Test - Testbench

Links ist die Toplevel-Schaltung zu sehen. Neben dem Clock und dem Reset weist der Block die zwei Eingangssignale der beiden RS-232 Ports auf. Ebenfalls bekommt er das Status-Signal STS vom FLASH Speicher zugewiesen.

Die Ausgänge dieses Blockes sind die Anschlüsse, die hardwaremässig mit den FPGA Pins gemappt werden, d.h. das z.B. die Kontrollsignale Chip Enable, Write Enable und Output Enable bereits aktiv tief sind. Natürlich sind auch die Adress- und Datenbusse mit dem Speicher verbunden. Ausserdem sind die Debug Ausgänge wie die LEDs oder die mezzaIRQ Signale zu sehen. Diese können auf Wunsch in der Simulation angezeigt werden.

Für die Speichersimulation wurde der *flash* Block verwendet. Dieser wurde von der HES-SO Wallis zur Verfügung gestellt und simuliert den verbauten Intel FLASH Speicher bis auf kleine Details genau. Die Betriebsmodi zum Löschen, Schreiben und Lesen werden dank der internen Zustandsmaschine gut simuliert. Es kann zudem eine .srec Datei definiert werden. Diese Datei muss sich im *work* Verzeichnis befinden und dient als Speicherinhalt während der Simulation. Wie zu sehen ist, wird das cen Signal an CE(0) angeschlossen. Die restlichen CE Signale sowie das BYTE_n werden im Tester fix gesetzt, so wie sie auch auf dem Board verkabelt sind. Auch die Timings des Speichers sind exakt modelliert. Einzig die Block Erase Dauer wurde für die Simulation stark verkürzt.

Der Tester befindet sich im blauen Block. Er definiert den Clock, Reset, fixe Signale und generiert eine Sequenz von RS-232 Werten. Diese Werte umfassen Kontroll- und Datenwerte und werden im VHDL character Format gesendet, dies entspricht dem ASCII Format. Der Tester simuliert das Perl Script und hat einen ähnlichen Ablauf.

1. ERASE_COMMAND
2. START_ADDR_COMMAND, anschliessend Start Adresse
3. LENGTH_COMMAND, anschliessend Länge
4. WRITE_COMMAND
5. START_ADDR_COMMAND, anschliessend Start Adresse
6. LENGTH_COMMAND, anschliessend Länge
7. READ_COMMAND

Der VHDL Code des Testers ist in Beilage 22 zu sehen.

Nun werden alle durchgeführten Tests genau beschrieben. Ebenfalls werden die Intel Flash Timings anhand der Simulation erklärt.

6.2.1.1 Block Erase Befehl Folgende Simulation zeigt den ersten Zyklus des Block Erase Vorgangs auf. Auf dem rs232_0data Signal kommt vom Tester der ASCII Wert x, daraufhin wird vom Control Extractor das Signal erasecommand aktiviert. Dies aktiviert über Zwischensignale, welche nicht dargestellt sind, die FSM und diese startet ihren Block Erase Ast. D.h. es wird bei Adresse 0x000000 mit dem Löschen des Speichers begonnen. Pro Block sind zwei Schreibzugriffe nötig, einen Löschbefehl (0x20) und eine Löschbestätigung (0xD0). Mehr Informationen dazu sind im Intel Datenblatt Seite 18 zu finden. Diese Befehle werden in Form von Daten über das Signal data an den Speicher übermittelt. Da es sich um Schreibzugriffe handelt, werden die Signale cen und wen aktiv tief bei den beiden Zugriffen. Informationen zu den genauen Timings dieser Abläufe folgen in Kürze.

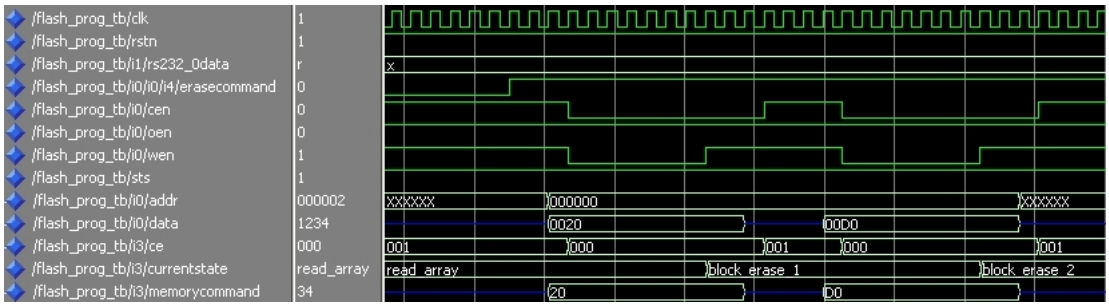


Abbildung 88: FLASH Test - Block Erase - 1. Zyklus

Nachdem der Speicher die Löschbestätigung erhalten hat, bereitet die interne Zustandsmaschine den Vorgang vor und führt ihn durch. Während dieser Zeit wechselt das STS Signal auf busy ('0'). Dies ist in untenstehender Abbildung erkennbar. Erst sobald der FLASH den Block gelöscht hat, wechselt das STS Signal zurück auf ready ('1'). Anschliessend wird der nächste Block gelöscht.

Der Vorgang wiederholt sich solange, bis der letzte Block mit Startadresse 0x1F0000 erreicht wird. Nachdem dieser gelöscht wurde, ist der Block Erase Vorgang abgeschlossen. Dies ist ebenfalls in untenstehender Abbildung zu sehen.

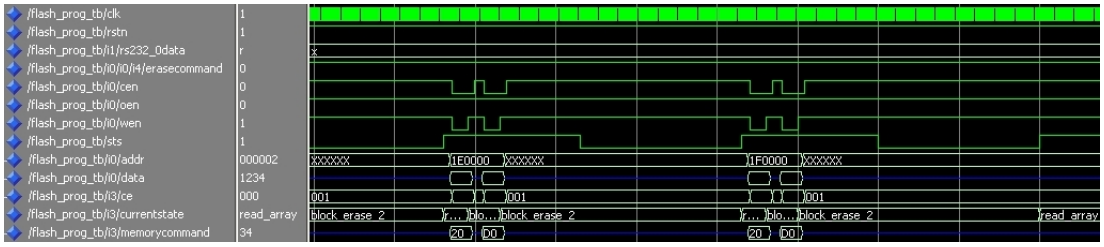


Abbildung 89: FLASH Test - Block Erase - Letzter Zyklus

Am Ende des Löschvorgangs wird eine Löschbestätigung über RS-232 zurück ans Perl Script geschickt. Dies ist allerdings nicht zu sehen. Die Bestätigung funktioniert immer gleich, daher wird sie nur im Kapitel zur Write Simulation aufgezeigt.

6.2.1.2 Start Address Befehl Dieser Vorgang wird mit dem ASCII Charakter s gestartet, welcher über rs232_0data vom Tester versandt wird. Daraufhin wird das startaddrcommand Signal aktiv und der Control Extractor sammelt die nächsten sechs Werte. Es sind immer sechs Werte, da das Perl Scripts jede eingegebene Start Adresse in ein sechsstelliges Format bringt. Das Einsammeln ist auf dem Signal rxsixnibbles zu sehen. Sobald sechs Werte angekommen sind, macht der counter6 einen Überlauf und das Signal counterof6 wird kurz aktiv. Daraufhin wird vom Control Extractor der Wert von rxsixnibbles ins Register startaddr kopiert. Die Start Adresse steht nun der FSM zur Verfügung. Die untenstehende Simulation bestätigt diesen Ablauf.

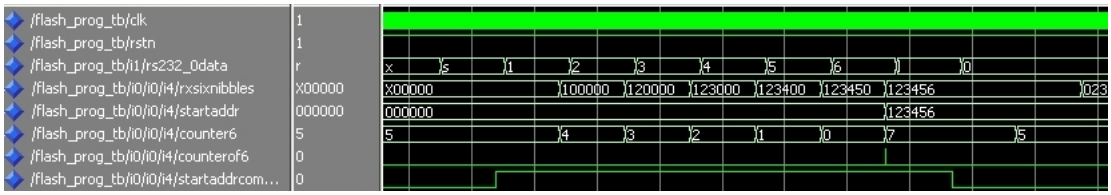


Abbildung 90: FLASH Test - Start Address

6.2.1.3 Length Befehl Analog zum Start Address Befehl hier nun der Length Befehl. Gestartet wird dieser durch ein l auf dem RS-232 Eingang. Der Control Extractor aktiviert das Signal lengthcommand und sammelt die nächsten vier Werte auf rxfournibbles ein. Auch hier wird die Länge immer durch das Perl Script auf vier fixiert. Sobald counter einen Überlauf macht, wird das Signal counterof aktiv und veranlasst ein Kopieren des rxfournibbles Signals in das Register length. Der Längenwert steht nun der FSM zur Verfügung.

Abbildung 91: FLASH Test - Length

6.2.1.4 Write Befehl Ein Schreibvorgang wird durch den Charakter w gestartet. Daraufhin arbeitet der Control Extractor einen ähnlichen Ablauf ab wie beim Length Befehl. Nur wird der Wert von rxfournibbles nun nach flashData kopiert. Dies ist in untenstehender Abbildung zu sehen.

Ähnlich wie beim Block Erase sind beim Write Befehl ebenfalls zwei Schreibzugriffe auf den Speicher nötig. Erst muss dem Speicher signalisiert werden, dass nun ein Schreibvorgang eingeleitet werden soll. Dies wird mit dem Befehl 0x40 über das data Signal

gemacht. Daraufhin ändert die interne Zustandsmaschine im Speicher ihren Zustand zu Word Programm. Bei einem Befehl können die acht höherwertigen Bits den Wert Don't Care ('-') haben. Während dem zweiten Zugriff können nun die zu schreibenden Daten über data übermittelt werden. Die Daten werden von der FSM aus dem Register flashData bezogen. Das data Signal wird bei Nichtverwendung stets auf hohe Impedanz 'Z' gezogen. Die Adresse muss nur während dem zweiten Zugriff die gewünschte Zieladresse sein, jedoch wurde in diesem Fall die Adresse während dem ganzen Vorgang direkt korrekt gesetzt. Natürlich werden die Kontrollsignale cen und wen gemäss den Intel Timings gesetzt.

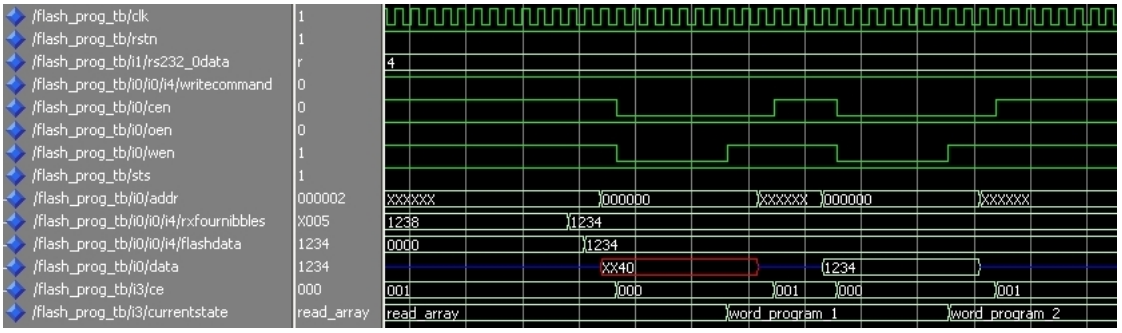


Abbildung 92: FLASH Test - Write - ein Zyklus

Folgende Abbildung zeigt mehrere Schreibvorgänge auf. Es ist zu sehen wie durch ein w der Schreibvorgang gestartet wird (writecommand aktiv) und wie sich das rxfournibbles Signal füllt und nach flashData kopiert wird. Das FLASH Modell in der Simulation wechselt nach jedem Schreibvorgang zurück in den Read Array Modus. Dies ist in der Hardware nicht der Fall, stört aber während der Simulation nicht. Ebenfalls ist das Toggeln des STS Signals erkennbar. Auf dieses wartet die FSM immer, allerdings ist die RS-232 Übertragung deutlich langsamer als ein Schreibvorgang im FLASH.

Auch der Fortschritt des lengthfsm Signal ist gut zu sehen. Die Anzahl geschriebener Charakter wird korrekt abgezogen.

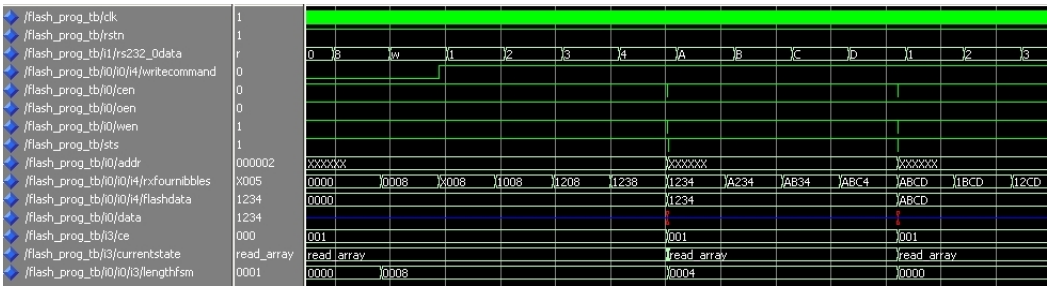


Abbildung 93: FLASH Test - Write - kompletter Vorgang

Da die Länge nach dem Schreiben von ABCD nun '0' beträgt, ist für die FSM der Schreib-

zugriff beendet. Es kann nun eine Bestätigung über RS-232 an das Perl Script gesendet werden. Dies ist in untenstehender Abbildung zu sehen. Über txdata1 wird erst ein v und dann ein CR (carriage return, leider von ModelSim als leerer Wert dargestellt) übermittelt. Während der Übermittlung wird dem RS-232 Block durch txwr1 das Eintreffen eines neuen Wertes signalisiert. Da txfull1 während dieser Zeit immer auf '0' ist, ist der FIFO Buffer nicht voll und ein Senden ist möglich.

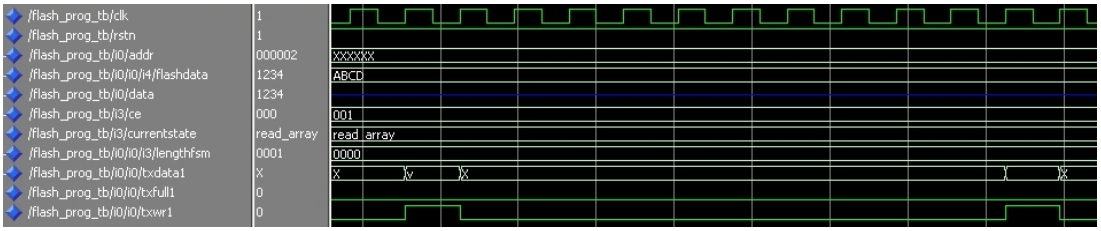


Abbildung 94: FLASH Test - Write - Bestätigung

Analog dazu werden alle anderen Bestätigungssignale an den Host verschickt.

6.2.1.5 Read Befehl Ein Lesen wird durch den Charakter r gestartet. Zuvor werden nochmals, wie oben beschreiben, ein Start Address und Length Befehl ausgeführt. Dies um der FSM für den Lesevorgang nötigen Registerwerte zu setzen.

Bei einem Lesen muss der Speicher wieder in den Read Array Modus gesetzt werden. In der Simulation ist dies automatisch der Fall, in der Hardware allerdings nicht. Dazu ist ein Schreibzugriff mit dem Befehl 0xFF bei irgendeiner Adresse nötig. Die Adresse wurde in diesem Fall mit der Start Adresse des Lesevorgangs gleichgesetzt. Sehr wichtig ist zu bemerken, dass bei diesem Zugriff das STS Signal nicht toggelt. Die FSM wartet also nicht auf eine Änderung des STS Signals, sondern beginnt direkt mit dem Lesen der ersten Adresse. Dies ist in untenstehender Abbildung gut erkennbar.

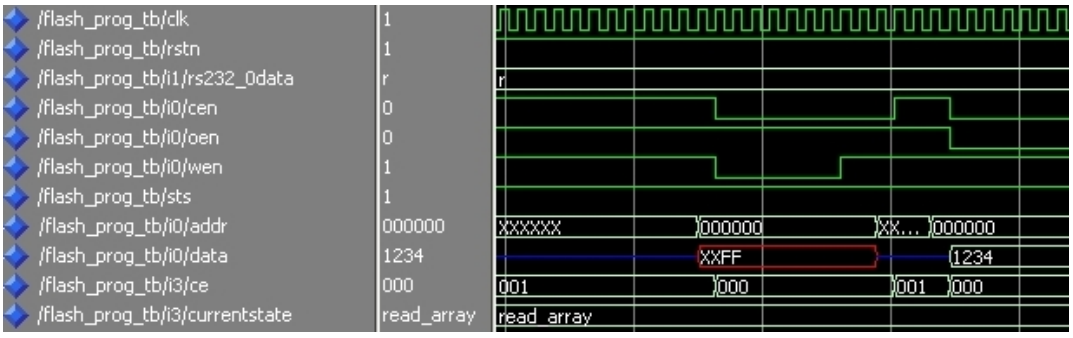


Abbildung 95: FLASH Test - Read - ein Zyklus

Der gelesene Wert ist ein 16 Bit Wert und enthält somit vier HEX Werte. Diese werden

nun von der FSM Wert für Wert extrahiert und in einen 8 Bit ASCII Wert umgewandelt. Der ASCII Charakter entspricht dabei weiterhin dem HEX Charakter.

In untenstehender Abbildung ist dies zu sehen. Der erste Nibble mit dem Wert A wird umgewandelt und über txdata1 an den Host geschickt.

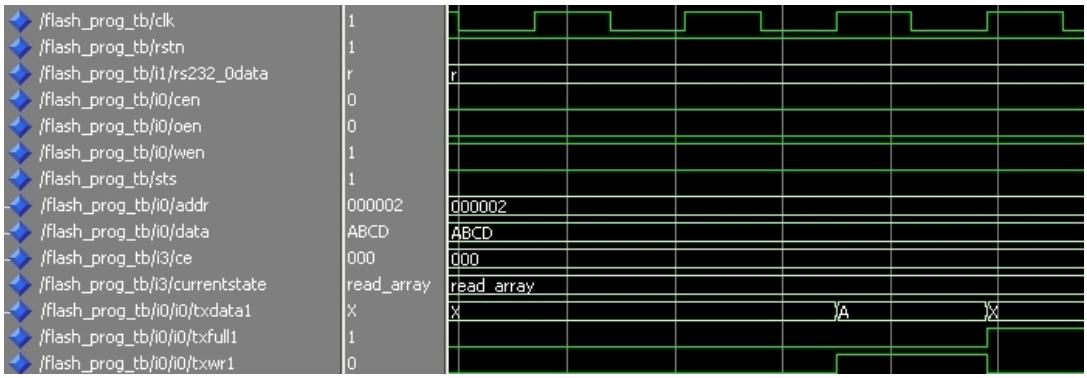


Abbildung 96: FLASH Test - Read - nach RS-232

6.2.1.6 Intel Flash Timings Die Zugriffe auf den FLASH Speicher müssen die von Intel definierten Timings einhalten. Je nach Zugriff oder Operation gibt Intel Maximal- oder Minimalwerte an. Im Falle der FLASH Schaltung sind die Timings beim Schreibzugriff wichtig. Beim Lesezugriff verlangsamt das RS-232 Interface den Vorgang so stark, dass die FLASH Timings ohne Probleme eingehalten werden können.

Daher werden nun auch nur die Write Timings erklärt und kontrolliert.

Ein Schreibzugriff wird dem Speicher durch eine '0' auf den Signalen Chip Enable (cen) und Write Enable (wen) signalisiert. Beide Signale sind aktiv tief und können gleichzeitig aktiviert werden. Erfasst werden die Daten sobald das wen Signal wieder auf '1' geht (steigende Flanke). Vor diesem Zeitpunkt muss die Zieladresse sowie die Daten an sich auf den entsprechenden Bus gelegt werden. Intel schreibt für die Adressen eine Mindestzeit von 55ns (W5) vor, für die Daten 50ns (W3). Ein Write Pulse mit dem wen Signal muss allerdings mindestens 70ns (W3) dauern. Das heisst, dass wenn die Adressen und Daten zeitgleich mit dem cen und wen Signal bereit gelegt werden, der Write Pulse W3 die längste Wartezeit ist. Dies ist in dieser Schaltung der Fall und in untenstehender Grafik ist ein Wert von 98ns gemessen worden. Somit ist dieses Timing erfüllt.

Um einen Schreibzugriff zu beenden, muss nach dem Entfernen von wen noch cen wieder auf '1' gesetzt werden. Intel schreibt hier eine Dauer von mindestens 10ns (W6) vor. In der Simulation wird ein Wert von 42ns gemessen, das Timing ist erfüllt. Die Adressen und Daten dürfen direkt mit wen wieder entfernt werden.

Die Dauer zwischen zwei Schreibzugriffen wird von Intel mit 30ns (W9) angegeben. Gemessen werden 98 ns, auch dieses Timing ist erfüllt.

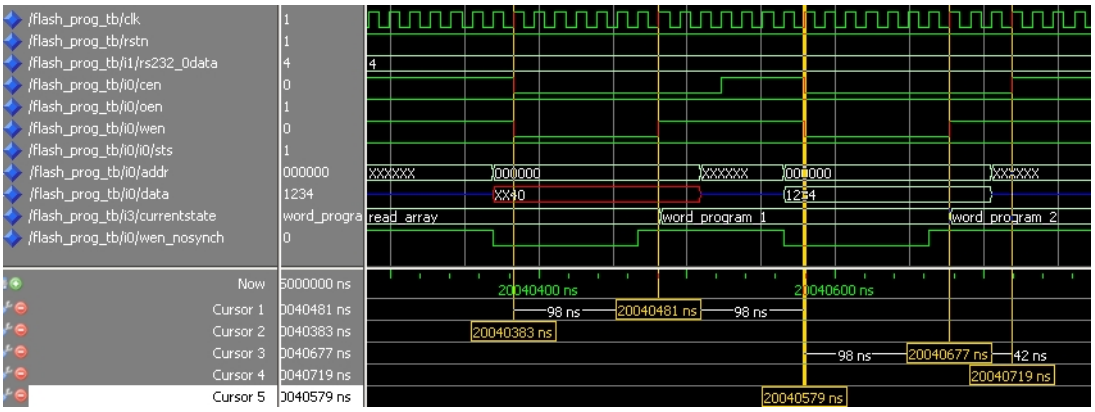


Abbildung 97: FLASH Test - Intel FLASH Write Timings

Eine genaue Auflistung aller Timings inkl. Waveform ist dem Intel Datenblatt zu entnehmen. Ab Seite 51 für Lesezugriffe und ab Seite 53 für Schreibzugriffe.

Die Wartezeiten wurden, wie bereits erwähnt, mit Waitstates in der FSM realisiert. Diese basieren auf einer Clockperiode und es lassen sich die zu wartenden Perioden definieren.

6.2.2 Perl Scripts und Funktionalitätstests

Mit Hilfe der Perl Scripts kann nicht nur die Schaltung bedient werden, sondern auch getestet werden. Bei diesem Test werden sowohl die Perl Scripts wie auch das gesamte System getestet.

Es wird mit dem Lösch-Script zuerst der ganze Speicher gelöscht, anschliessend wird eine Test .srec Datei mit dem Write-Script in den FLASH geladen und zuletzt wird der Speicher bei verschiedenen Adressen mit dem Read-Script gelesen. Die generierte .srec Datei kann nun mit der Test .srec Datei verglichen werden um ein korrektes Funktionieren des Systems zu bestätigen.

Folgende .srec Datei wird zum Test verwendet:

flash.srec

```
S0110000666C6173687465737 42E7372656345
S315000000008810000009000000581C120B00100000031
S315000000010A1480000A7500000108007 12AC10200174
S3150000002091D020000100000001000000010000046
S3150000003091D020000100000001000000010000036
S315000000040A1480000290000 0681C523980100000090
S315000000050A1480000290000 0581C520FC0100000020
S315000000060A1480000290000 0581C52168010000 0A3
S31540000A300000000000000000000000000000000070
S31540000A400000000000000000000000000000000060
S31540000A500000000000000000000000000000000050
S31540000A600000000000000000000000000000000040
S31540000A700000000000000000000000000000000030
S31540000A800000000000000000000000000000000020
S705000000000FA
```

Abbildung 98: FLASH Test - flash.srec Testdatei

Die Datei beinhaltet S0 und S3 Records. Ebenso ist eine fixe Adressierung vorgegeben. Die Adressen zeigen bei den ersten Records in den FLASH Bereich, später folgen Records für den RAM Bereich.

Nach einem Lesen des Speichers bei Adresse 0x0 wird folgende .srec Datei erstellt:

flashReadout.srec

```
S0XXFollowing Data read out from FLASH
S315000000008810000009000000581C120B001000000XX
S315000000010A1480000A7500000108000712AC102001XX
S3150000002091D020000100000000100000001000000XX
S3150000003091D020000100000000100000001000000XX
S315000000040A1480000290000000681C5239801000000XX
S315000000050A1480000290000000581C520FC01000000XX
S315000000060A1480000290000000581C52168010000000XX
S315000000070FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFXX
S315000000080FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFXX
S315000000090FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFXX
```

Abbildung 99: FLASH Test - flashReadout.srec - gerade Adresse (0x0)

Dieser Test bestätigt, dass das System korrekt funktioniert. Alle Datenwerte stimmen 1:1 mit der Quelldatei flash.srec überein. Die erste Zeile ist wie gewünscht ein individueller S0 Record und bei den S3 Records stimmt die Adressierung ebenfalls. Die Checksum jedes Records wurde nicht errechnet, hier sind die Platzhalter zu sehen.

Da mehr Records gelesen als geschrieben wurden, sind die restlichen gelesenen Datenwerte alle 0xF. Der Intel Speicher setzt beim Löschen alle Bits auf '1'. Somit wird auch der Löschvorgang korrekt durchgeführt. Dieser Test zeigt auch auf, dass keine Records mit RAM Adressbereich geschrieben wurden.

Beim nächsten Test wird der Lesevorgang bei Adresse 0x3 begonnen. Folgende Datei wird generiert:

flashReadout.srec

```
S0XXFollowing Data read out from FLASH
S315000000030000090000581C120B001000000A148XX
S3150000001300000A7500000108007 12AC10200191D0XX
S315000000232000010000000100000001000000091D0XX
S315000000332000010000000100000001000000A148XX
S315000000430000290000 0681C5239801000000A148XX
S315000000530000290000 0581C520FC01000000A148XX
S315000000630000290000 0581C52168010000000FFFFXX
S31500000073FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFXX
S31500000083FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFXX
S31500000093FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFXX
```

Abbildung 100: FLASH Test - flashReadout.srec - gerade Adresse (0x3)

Der Aufbau der Datei ist wie beim ersten Test und somit korrekt. Die Adressierung beginnt nun bei Adresse 0x3 und inkrementiert in den korrekten 16er Sprüngen. Beim Lesen werden die Daten nun versetzt in die .srec Datei gespeichert. Die ersten zwei Bytes werden übersprungen und beim dritten Byte, was auch der Startadresse entspricht, werden die Daten gelesen. Dieser Offset ist auch beim letzten gelesenen Record wieder zu sehen. Somit funktioniert das System auch, wenn bei ungeraden Adressen gelesen wird.

6.2.3 Schlussfolgerung

Die Schaltung zur FLASH Programmation funktioniert wie geplant. Der Speicher kann gelöscht, geschrieben und gelesen werden.

Ein Problem ist allerdings noch vorhanden. Nach dem Lesevorgang muss die Schaltung per Reset Button zurück gesetzt werden. Da die Arbeiten an der LEON Schaltung eine höhere Priorität hatten, wurde dieser kleine Fehler noch nicht behoben.

6.2.4 Verbesserungsmöglichkeiten

Folgende Punkte sind bei der FLASH Schaltung für zukünftige Versionen zu verbessern:

- Sicherheitsmassnahmen bei der RS-232 Übertragung, z.B. Parity Checks
- höhere RS-232 Baudrate
- Fehler nach Lesevorgang beheben

7 Arbeitsjournal

Dieses Kapitel gibt einen Überblick über den Ablauf der Diplomarbeit. Es wird für jede Woche kurz zusammengefasst welche Arbeiten durchgeführt wurden und welche Probleme aufgetaucht und gelöst worden sind.

Arbeitsjournal Diplomarbeit	
Woche	**Durchgeführt**
Frühlingssemeter 1 Tag pro Woche	Installation der Arbeitsstation inkl. Entwicklungs- und Dokumentationssoftware Einlesen in Dokumentationen von Gaisler zu GRLIB, Entwicklungstools und IP Cores Einlesen in Dokumentationen von AMBAdraw inkl. ersten Tests und Versuchen Einarbeitung in Perl und Serielle Kommunikation Beginn des LEON System Entwurfs, Auswahl der IP Cores, Generics Parameter analysiert und erster Memory Map Entwurf
1 *11.05. - 17.05.09*	Memory Map und Adressierung beendet AMBAdraw Projekt erstellt Generics konfiguriert Export in HDL Designer LEON Schaltung durch Pads und Toplevel erweitert
2 *18.05. - 24.05.09*	LEON Schaltung durch Pads und Toplevel erweitert (Fortsetzung) Simulation und Tests, Speicher (FLASH, SDRAM) Simulation. Problematische FLASH Simulation, Teillösung gefunden durch Modifizieren des Gaisler SRAM Blocks *Bemerkung: Do, 21.05.09 Feiertag*
3 *25.05. - 31.05.09*	Beginn Place & Route der LEON Schaltung, unterbrochen durch FPGA Tech-Fehler bei Ethernet IP Core, noch keine Lösung gefunden Entwurf und Beginn Realisation der FLASH Programmierungsschaltung, Übertragungsprotokoll definiert, RS-232 Schnittstelle, Daten/Kontroll Filter und Zustandsmaschine implementiert Simulation und Tests der FLASH Schaltung Perl Script zum Einlesen und Versenden der Daten entwickelt Place & Route der FLASH Schlatung
4 *01.06. - 07.06.09*	Krankheitsbedingter Ausfall

Tabelle 23: Arbeitsjournal Woche 1 bis 4

Arbeitsjournal Diplomarbeit	
Woche	**Durchgeführt**
5 *08.06. - 14.06.09*	Fortsetzung der Arbeiten an der FLASH Schaltung Überarbeiten des Übertragungsprotokolls und Anpassung des Filters, der Zustandsmaschine und des Perl Scripts Weitere Simulationen inkl. FLASH Simulation und Read/Write Vorgängen Place & Route der LEON Schaltung fortsetzen, Tests durch Ändern der FPGA Techs, temporäres Deaktivieren des Ethernet IP Cores usw. Noch keine Lösung gefunden Simulation einer Demo-Schaltung von Gaisler zur Kontrolle der Speicherfunktionalität *Bemerkung: Mo, 08.06.09 Krankheitsbedingter Ausfall,* *Do, 11.06.09 Feiertag*
6 *15.06. - 21.06.09*	FLASH Schaltung Simulation mit genauem Modell des Speichers, Timinganpassungen an der FSM vorgenommen FLASH Schaltung auf FPGA implementiert und getestet Lösung für das Synthese/Place & Route Problem der FPGA gefunden. LEON Schaltung P&R beendet.
7 *22.06. - 28.06.09*	LEON Schaltung Kapazitätsproblem analysiert, Schaltungsumfang minimiert, Ethernet deaktiviert USB IP Core Spezifikation erstellt jedoch zu wenig FPGA Kapazität für eine Umsetzung LEON Schaltung mit FLASH und SDRAM Modellen simuliert inkl. Konfiguration des Speichercontrollers über Software
8 *29.06. - 06.07.09*	Weitere Tests der LEON Schaltung zur Ansteuerung von FLASH und SDRAM *Projekt Deadline 29.06. 18:00* Beenden der technischen Dokumentation 01.07. Abgabe des Dossiers an eine Druckerei (2 Tage) 06.07. Abgabe Diplomarbeit

Tabelle 24: Arbeitsjournal Woche 5 bis 8

Die Dokumentation wurde stets nebenbei erstellt. Sobald eine Aufgabe beendet wurde, wurde das entsprechende Kapitel in diesem Dokument verfasst. Die ausgefallene Woche kann nicht kompensiert werden, da das Abgabedatum fix eingehalten werden muss.

8 Schlussfolgerung

Ein LEON Prozessorsystem wurde auf dem FPGA_EBS_V2.0 Board implementiert. Das System musste aus FPGA Kapazitätsgründen auf ein Minimum reduziert werden. Es funktioniert in der Simulation und auf der Hardware, allerdings sind zur Zeit nur 16 statt 32 Bit Zugriffe auf den SDRAM möglich. Ebenfalls wurde ein System zum Programmieren des FLASH Speichers entwickelt, welches seine Aufgabe zuverlässig erfüllt.

Im Rahmen dieser Diplomarbeit konnten allerdings nicht alle Aufgaben erfüllt werden. Dies hatte teils technisch bedingte Gründe wie beim USB IP Core oder es fehlte die Zeit dafür, da viele zeitaufwendige Probleme gelöst werden mussten und zudem leider eine Woche ausgefallen ist. Die Entwicklung der Demoapplikation wird bis zur Präsentation noch nachgeholt. Die fertig gestellten Arbeiten sind aber alle gründlich getestet und funktionieren wie gewünscht.

Die möglichen Verbesserungsvorschläge für zukünftige Ausbaustufen der entwickelten Systeme sind bereits erwähnt worden. Die Hauptproblematik war die Konfiguration der Gaisler IP Cores, vor allem für die korrekte Speicherverwaltung. Dort sind, wie erwähnt, noch einige Punkte ausstehend. AMBAdraw bietet hier eine sehr benutzerfreundliche Bedienung an und erstellt eine übersichtliche Konfigurationsdatei, jedoch sind die Parameter von Gaisler zumeist nur sehr oberflächlich dokumentiert. So musste vieles mit Simulationen getestet werden. Dies nahm sehr viel Zeit in Anspruch.

Als persönliche Schlussbemerkung möchte ich anfügen, dass es sehr interessant war, ein FPGA System selber zu erstellen. In den Labors, während der Studienzeit, hat man zwar gelernt wie eine FPGA programmiert wird, einen Einblick in deren Implementation hat man allerdings nur sehr oberflächlich erhalten. Aus diesem Grunde habe ich mich für diese Diplomarbeit entschieden. Der Mix aus Hardware- und Softwareentwicklung ist für mich faszinierend. Während der Diplomarbeit habe ich nun einen detaillierten Einblick in die FPGA Hardwareentwicklung erhalten und habe gesehen, welche zusätzlichen Etappen zu den Labors noch dazu kommen.

Erstaunt war ich allerdings wie enttäuschend die Dokumentation von Gaisler ist. Die Datenblätter sind allesamt sehr oberflächlich gehalten und die Implementierung ihrer Cores sehr verschachtelt und nicht kommentiert. Bei Problemen dauert es so umso länger bis eine Lösung gefunden werden kann. Da weicht die Praxis sehr stark von der erlernten Theorie ab.

Zum Abschluss möchte ich mich bei den Personen bedanken, die mir während der Diplomarbeit stets hilfsbereit zur Seite gestanden sind:

- François Corthay, Dozent und Experte dieser Diplomarbeit
- Oliver Gubler, technischer Mitarbeiter
- Silvan Zahno, technischer Mitarbeiter
- Sébastien Farquet, technischer Mitarbeiter

9 Beilagen

Beilage 1 Pflichtenheft
Beilage 2 GRLIB & AMBA Bus Implementation
Beilage 3 LEON3 GRLIB Auszug
Beilage 4 AHBCTRL GRLIB Auszug
Beilage 5 APBCTRL GRLIB Auszug
Beilage 6 DSU3 GRLIB Auszug
Beilage 7 MCTRL GRLIB Auszug
Beilage 8 GRETH GRLIB Auszug
Beilage 9 AHBUART GRLIB Auszug
Beilage 10 APBUART GRLIB Auszug
Beilage 11 IRQMP GRLIB Auszug
Beilage 12 GRGPIO GRLIB Auszug
Beilage 13 config.vhd - VHDL Generics
Beilage 14 toplevel_arch_struct.vhd - IP Cores & I/Os
Beilage 15 VHDL Code der Pad Blockschemas
Beilage 16 LEON Schaltung .ucf (I/O Mapping)
Beilage 17 FLASH Programmation Toplevel
Beilage 18 FLASH Control Extractor VHDL Architektur
Beilage 19 FLASH Perl Scripts
Beilage 20 FLASH Programmation .ucf (I/O Mapping)
Beilage 21 LEON Testapplikation readtest.c
Beilage 22 VHDL Code FLASH Tester

Der Inhalt und die Struktur der CD sind der *readme.txt* zu entnehmen.

10 Quellen und Links

- Gaisler GRLIB Dokumentation, grlib.pdf und grip.pdf
- Gaisler GRMON Dokumentation, grmon.pdf
- Intel FLASH Speicher Dokumentation, flash - 28F320J3A.pdf
- Micron SDRAM Speicher Dokumentation, sdram - MT48LC8M16A.pdf
- Cypress USB Controller Dokumentation, usb - fx2.pdf
- FPGA Board Schema, schematics.pdf
- Bericht von Herrn Valentini, Valentini_TD_Bericht.pdf
- Bilder Kapitel 4.1 LEON System aus Gaisler grlib.pdf und grip.pdf
- Bilder Kapitel 4.2 USB Controller Core aus Cypress FX2 Datasheet usb - fx2.pdf
- Bilder Kapitel 5.1.6 SDRAM aus Micron Datasheet sdram - MT48LC8M16A.pdf
- Gaisler Support Group unter http://tech.groups.yahoo.com/group/leon_sparc/
- Wikipedia Online Enzyklopädie unter http://wikipedia.org/

Bis auf die Online Links sind alle Quellen auf der CD enthalten.

Sion, den 06 Juli 2009

Nanzer Thomas

Beilage 1

Pflichtenheft

Prozessorsystem auf FPGA

—

Pflichtenheft

Diplomarbeit 2009

Autor:	Nanzer Thomas
Projekt:	Prozessorsystem auf FPGA
Experten:	François Corthay, Prof. Ivan Defilippis
Datum:	13. März 2009
Version:	1.0

Inhaltsverzeichnis

1 Einleitung

Dieses Dokument beschreibt den genauen Funktionsumfang und das Ziel des Projekts.Es ist wichtig für die Kontrolle am Ende des Projekts, ob alle Zielsetzungen erfüllt wurden.

2 Pflichtenheft

2.1 Zielbestimmungen

Musskriterien

- Prozessorsystem auf Basis eines LEON- und AMBA Systems mit Hilfe von AMBArchitect entwickeln
- Verwendung des FPGA Boards FPGA_EBS_V2.0 der HES-SO//Wallis mit folgenden Komponenten:
  - FPGA Xilinx Spartan-3EXC3S500E
  - Flash Intel 28F320J3A
  - SDRAM Micron MT48LC8M16A2
  - Ethernet
  - Slave USB
  - JTAG
  - RS-232
- Mikroprozessorsystem-Prototyp in der FPGA erstellen
- System mit Hilfe einer Debug Support Unit (DSU) auf fehlerlose Operation überprüfen
- Multi-Plattform kompatibler Debug-Interface Monitor entwickeln für:
  - Programmation des FLASH Speichers
  - Debugfunktionen:
    - Start/Stop Program
    - Read/Write Memory
- Ethernet und/oder USB Verbindung testen

Wunschkriterien

- Anwendung zu Demonstrationszwecken entwickeln
- Individualisierung der USB ID Erkennung
- Programmation FLASH per USB oder RS-232 Schnittstelle

2.2 Produktübersicht

Die auf dem HES-SO//Wallis FPGA Board verfügbaren Komponenten werden zusammen in einem Mikrocontrollersystem kombiniert. Zentrales Element dieses Mikrocontrollers ist der in der FPGA implementierte Mikroprozessor. Dieser basiert auf einem LEON3 Prozessor welcher mit seiner Peripherie über den AMBA Bus kommuniziert. An diesem multiplexten Master/Slave Bus sind, neben dem Prozessor, alle Schnittstellen (USB, Ethernet, usw.), Debug-Schnittstelle als auch der Speicher (Flash, SDRAM) angeschlossen.
All diese Elemente können in Blöcken, sogenannten IP Cores, von Aeroflex Gaisler[1] bezogen werden. Ein Grossteil der IP Cores sind im VHDL Format unter der GNU GPL Lizenz verfügbar. Jedoch ist u.a. die Debug-Konsole unter kommerzieller Lizenz zu beziehen. Deshalb wird im Rahmen dieser Diplomarbeit ein Multi-Plattform kompatibler Debug-Interface Monitor entwickelt. Mit Hilfe dieses Monitors lässt sich anschliessend die Kommunikation zur Debug Schnittstelle bedienen.
Zu Demonstrationszwecken wird auf dem implementierten Mikrocontrollersystem eine Anwendung entwickelt, welche einige der verbauten Komponenten verwendet und ein Funktionieren des Systems aufzeigt.

2.3 Produktfunktionen

LEON3 Microprozessor

In der FPGA wird der von Gaisler zur Verfügung gestellte LEON IP Core implementiert. Es handelt sich um einen 32-Bit, 7 Stage Pipeline Prozessor auf SPARC V8 Architektur. Der von Gaisler entwickelte Prozessor ist vielfältig konfigurierbar und eignet sich gut für SOC (system-on-a-chip) Lösungen. Er wird als Master an den AMBA Bus angeschlossen und kann mit allen Slave IP Cores am selben AMBA Bus kommunizieren.

[1] www.gaisler.com

AMBA Bus

Der AMBA Bus dient als Verbindungsträger aller Adress- und Datenkommunikationen des Systems. Er ist intern in zwei Busse aufgeteilt:

- AHB (Advanced High-Performance Bus): eingesetzt für schnelle Peripherie welche viel Datenverkehr und schnelle Zugriffszeiten verlangt.
- APB (Advanced Peripheral Bus): eingesetzt für langsamere Peripherie, ebenfalls sind die Konfigurationsregister der IP Cores über diesen Bus erreichbar.

Beide Unterbusse sind multiplexte Master-Slave Busse. Verbunden werden beide über eine Brücke. Gaisler stellt für den gesamten AMBA Bus alle nötigen Controller und Brücken bereit und erweitert diese mit Adressdekodierung, Interrupt Funktionen, Plug & Play Fähigkeiten usw.

Das System muss aber auch ohne diese Gaisler Zusatzfunktionen AMBA Bus konform bleiben. Daher erhalten beispielsweise alle IP Cores fixe Adressen, trotz der vorhandenen Plug & Play Fähigkeiten.

SDRAM/FLASH

Auf dem HES-SO//Wallis FPGA Board sind zwei Speicherbausteine verbaut, ein SDRAM und ein FLASH Speicher. Diese werden von Memory Controller IP Cores angesprochen. Diese Controller sind ebenfalls über den AMBA Bus mit dem Rest des Systems verbunden. Der FLASH Speicher enthält den Programmcode und wird bei Ausführung in den RAM kopiert. Der FLASH Speicher wird über die FPGA programmiert, sei es direkt per Gaisler IP Cores oder via temporärer Umprogrammierung des FPGA Bausteines.

USB/Ethernet

Die verbauten USB Schnittstellen werden von einem programmierbaren Cypress USB Controller verwaltet. Dieser stellt die Daten via FIFO Buffern (einen pro Endpoint) dem System bereit. Es wird kein Gaisler IP Core eingesetzt. Allerdings muss eine AMBA Bus Komponente erstellt werden, um den USB Controller mit den restlichen IP Cores zu verbinden.
Für die Ethernet Schnittstelle wird ein Gaisler IP Core Controller in der FPGA implementiert. Während des Projekts wird eine Anwendung realisiert, welche mindestens eine dieser Schnittstellen verwendet um deren Funktionalität aufzuzeigen. Diese Aufgabe wird voraussichtlich zusammen mit der Demo-Applikation realisiert.
Zudem können beide Schnittstellen als alternative Stromversorgung eingesetzt werden.

Debug-Funktionalität

Damit das FPGA Board für das Cross-Development inkl. umfangreichen Debug-Fähigkeiten ansprechbar ist, wird eine DSU (Debug Support Unit) eingesetzt. Diese ist als eigenständiger IP Core verfügbar und wird an den AMBA Bus angeschlossen. Die DSU verfügt aber über separate Leitungen zum LEON Core um diesen beispielsweise zu stoppen. Als Debug Schnittstelle kommt die verbaute RS-232 Schnittstelle zum Einsatz. RS-232 benötigt keine zusätzlichen Adapter und ist weit verbreitet. Die Schnittstelle wird ebenfalls über einen IP Core angesprochen.

Monitor für Debug-Schnittstelle

Wie bereits erwähnt, bietet Gaisler keinen Open Source Monitor an. Um auch für zukünftige Projekte einen Debug Monitor zu haben, wird während der Diplomarbeit ein solcher entwickelt. Dieser ist kommandozeilenbasiert und unterstützt ein Lesen und Schreiben der Register der verbauten Komponenten. Dabei wird die Scriptsprache PERL eingesetzt um sicherzustellen, dass der Monitor plattformunabhängig bleibt.

Demo-Applikation

Um das System zu testen und zu präsentieren wird eine Demo Applikation entwickelt.
In dieser Applikation wird ein Morse Code Sender und Empfänger System implementiert. In einer Windows-kompatiblen Applikation kann ein Satz eingetippt und verschickt werden. Über Ethernet und dem xPL Protokoll gelangen die Daten als Text-String zum LEON System, welches anhand des Morse Codes den Text übersetzt und an einer LED ausgibt. Ebenfalls ist es möglich an einem Taster des Boards einen Morse Code einzugeben und an die Applikation auf dem PC zu versenden. Auch hier wird die Umwandlung auf dem LEON System durchgeführt. In einer zweiten Phase wäre es möglich einen externen Lautsprecher, zusätzlich zur LED, anzuschliessen.

Sion, den 13. März 2009

Nanzer Thomas

_____

Beilage 2

GRLIB & AMBA Bus Implementation

5 GRLIB Design concept

5.1 Introduction

GRLIB is a collection of reusable IP cores, divided on multiple VHDL libraries. Each library provides components from a particular vendor, or a specific set of shared functions or interfaces. Data structures and component declarations to be used in a GRLIB-based design are exported through library specific VHDL packages.

GRLIB is based on the AMBA AHB and APB on-chip buses, which is used as the standard interconnect interface. The implementation of the AHB/APB buses is compliant with the AMBA-2.0 specification, with additional 'sideband' signals for automatic address decoding, interrupt steering and device identification (a.k.a. plug&play support). The AHB and APB signals are grouped according to functionality into VHDL records, declared in the GRLIB VHDL library. The GRLIB AMBA package source files are located in lib/grlib/amba.

All GRLIB cores use the same data structures to declare the AMBA interfaces, and can then easily be connected together. An AHB bus controller and an AHB/APB bridge are also available in the GRLIB library, and allows to assemble quickly a full AHB/APB system.

The following sections will describe how the AMBA buses are implemented and how to develop a SOC design using GRLIB.

5.2 AMBA AHB on-chip bus

5.2.1 General

The AMBA Advanced High-performance Bus (AHB) is a multi-master bus suitable to interconnect units that are capable of high data rates, and/or variable latency. A conceptual view is provided in figure 5. The attached units are divided into master and slaves, and controlled by a global bus arbiter.

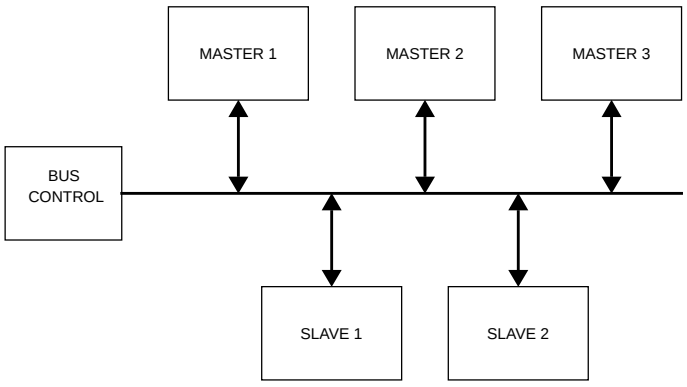


Figure 5. AMBA AHB conceptual view

Since the AHB bus is multiplexed (no tristate signals), a more correct view of the bus and the attached units can be seen in figure 6. Each master drives a set of signals grouped into a VHDL record called HMSTO. The output record of the current bus master is selected by the bus multiplexers and sent to the input record (ahbsi) of all AHB slaves. The output record (ahbso) of the active slave is selected by the bus multiplexer and forwarded to all masters. A combined bus arbiter, address decoder and bus multiplexer controls which master and slave are currently selected.

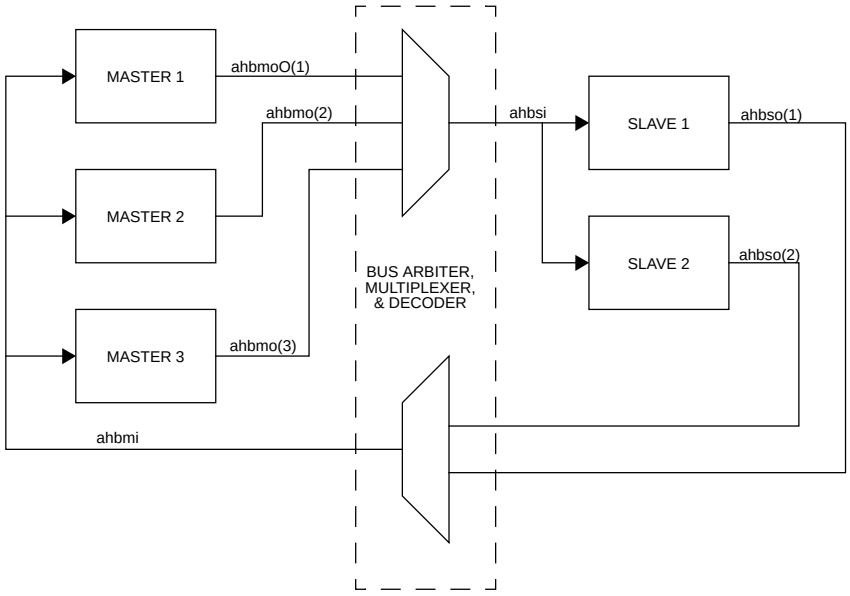


Figure 6. AHB inter-connection view

5.2.2 AHB master interface

The AHB master inputs and outputs are defined as VHDL record types, and are exported through the TYPES package in the GRLIB AMBA library:

```
-- AHB master inputs
  type ahb_mst_in_type is record
    hgrant  : std_logic_vector(0 to NAHBMST-1);       -- bus grant
    hready  : std_ulogic;                             -- transfer done
    hresp   : std_logic_vector(1 downto 0);           -- response type
    hrdata  : std_logic_vector(31 downto 0);          -- read data bus
    hcache  : std_ulogic;                             -- cacheable
    hirq    : std_logic_vector(NAHBIRQ-1 downto 0);   -- interrupt result bus
  end record;

-- AHB master outputs
type ahb_mst_out_type is record
    hbusreq           : std_ulogic;                       -- bus request
    hlock             : std_ulogic;                       -- lock request
    htrans            : std_logic_vector(1 downto 0); -- transfer type
    haddr             : std_logic_vector(31 downto 0); -- address bus (byte)
    hwrite            : std_ulogic;                       -- read/write
    hsize             : std_logic_vector(2 downto 0); -- transfer size
    hburst            : std_logic_vector(2 downto 0); -- burst type
    hprot             : std_logic_vector(3 downto 0); -- protection control
    hwdata            : std_logic_vector(31 downto 0); -- write data bus
    hirq              : std_logic_vector(NAHBIRQ-1 downto 0);-- interrupt bus
    hconfig           : ahb_config_type;            -- memory access reg.
    hindex            : integer range 0 to NAHBMST-1; -- diagnostic use only
  end record;
```

The elements in the record types correspond to the AHB master signals as defined in the AMBA 2.0 specification, with the addition of four sideband signals: HCACHE, HIRQ, HCONFIG and HINDEX. A typical AHB master in GRLIB has the following definition:

```
library grlib;
use grlib.amba.all;
library ieee;
use ieee.std_logic.all;

entity ahbmaster is
  generic (
    hindex : integer := 0);           -- master bus index
  port (
    reset   : in  std_ulogic;
    clk     : in  std_ulogic;
    hmsti   : in  ahb_mst_in_type;     -- AHB master inputs
    hmsto   : out ahb_mst_out_type     -- AHB master outputs
  );
end entity;
```

The input record (HMSTI) is routed to all masters, and includes the bus grant signals for all masters in the vector HMSTI.HGRANT. An AHB master must therefore use a generic that specifies which HGRANT element to use. This generic is of type integer, and typically called HINDEX (see example above).

5.2.3 AHB slave interface

Similar to the AHB master interface, the inputs and outputs of AHB slaves are defined as two VHDL records types:

```
-- AHB slave inputs
  type ahb_slv_in_type is record
    hsel      : std_logic_vector(0 to NAHBSLV-1);     -- slave select
    haddr     : std_logic_vector(31 downto 0);        -- address bus (byte)
    hwrite    : std_ulogic;                           -- read/write
    htrans    : std_logic_vector(1 downto 0);         -- transfer type
    hsize     : std_logic_vector(2 downto 0);         -- transfer size
    hburst    : std_logic_vector(2 downto 0);         -- burst type
    hwdata    : std_logic_vector(31 downto 0);        -- write data bus
    hprot     : std_logic_vector(3 downto 0);         -- protection control
    hready    : std_ulogic;                           -- transfer done
    hmaster   : std_logic_vector(3 downto 0);         -- current master
    hmastlock : std_ulogic;                           -- locked access
    hbsel     : std_logic_vector(0 to NAHBCFG-1);     -- bank select
    hcache    : std_ulogic;                           -- cacheable
    hirq      : std_logic_vector(NAHBIRQ-1 downto 0); -- interrupt result bus
  end record;

-- AHB slave outputs

  type ahb_slv_out_type is record
    hready     : std_ulogic;                            -- transfer done
    hresp      : std_logic_vector(1 downto 0);          -- response type
    hrdata     : std_logic_vector(31 downto 0);         -- read data bus
    hsplit     : std_logic_vector(15 downto 0);         -- split completion
    hcache     : std_ulogic;                            -- cacheable
    hirq       : std_logic_vector(NAHBIRQ-1 downto 0); -- interrupt bus
    hconfig    : ahb_config_type;                       -- memory access reg.
    hindex     : integer range 0 to NAHBSLV-1;          -- diagnostic use only
  end record;
```

The elements in the record types correspond to the AHB slaves signals as defined in the AMBA 2.0 specification, with the addition of five sideband signals: HBSEL, HCACHE, HIRQ, HCONFIG and HINDEX. A typical AHB slave in GRLIB has the following definition:

```
library grlib;
use grlib.amba.all;
library ieee;
use ieee.std_logic.all;

entity ahbslave is
  generic (
    hindex : integer := 0);           -- slave bus index
  port (
    reset   : in  std_ulogic;
    clk     : in  std_ulogic;
    hslvi   : in  ahb_slv_in_type;     -- AHB slave inputs
    hslvo   : out ahb_slv_out_type     -- AHB slave outputs
  );
end entity;
```

The input record (ahbsi) is routed to all slaves, and include the select signals for all slaves in the vector ahbsi.hsel. An AHB slave must therefore use a generic that specifies which hsel element to use. This generic is of type integer, and typically called HINDEX (see example above).

5.2.4 AHB bus control

GRLIB AMBA package provides a combined AHB bus arbiter (ahbctrl), address decoder and bus multiplexer. It receives the ahbmo and ahbso records from the AHB units, and generates ahbmi and ahbsi as indicated in figure 6. The bus arbitration function will generate which of the ahbmi.hgrant elements will be driven to indicate the next bus master. The address decoding function will drive one of the ahbsi.hsel elements to indicate the selected slave. The bus multiplexer function will select which master will drive the ahbsi signal, and which slave will drive the ahbmo signal.

5.2.5 AHB bus index control

The AHB master and slave output records contain the sideband signal HINDEX. This signal is used to verify that the master or slave is driving the correct element of the ahbso/ahbmo buses. The generic HINDEX that is used to select the appropriate hgrant and hsel is driven back on ahbmo.hindex and ahbso.hindex. The AHB controller then checks that the value of the received HINDEX is equal to the bus index. An error is issued dunring simulation if a missmatch is detected.

5.3 AHB plug&play configuration

5.3.1 General

The GRLIB implementation of the AHB bus includes a mechanism to provide plug&play support. The plug&play support consists of three parts: identification of attached units (masters and slaves), address mapping of slaves, and interrupt routing. The plug&play information for each AHB unit consists of a configuration record containing eight 32-bit words. The first word is called the identification register and contains information on the device type and interrupt routing. The last four words are called bank address registers, and contain address mapping information for AHB slaves. The remaining three words are currently not assigned and could be used to provide core-specific configuration information.

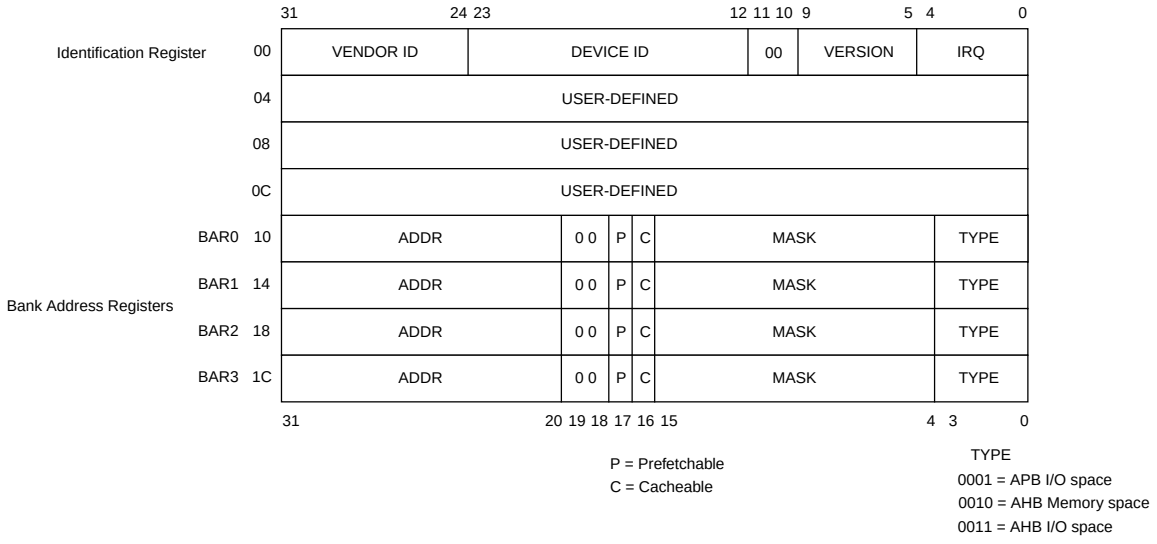


Figure 7. AHB plug&play configuration layout

The plug&play information for all attached AHB units appear as a read-only table mapped on a fixed address of the AHB, typically at 0xFFFFF000. The configuration records of the AHB masters appear in 0xFFFFF000 - 0xFFFFF800, while the configuration records for the slaves appear in 0xFFFFF800 - 0xFFFFFFFC. Since each record is 8 words (32 bytes), the table has space for 64 masters and 64 slaves. A plug&play operating system (or any other application) can scan the configuration table and automatically detect which units are present on the AHB bus, how they are configured, and where they are located (slaves).

The configuration record from each AHB unit is sent to the AHB bus controller via the HCONFIG signal. The bus controller creates the configuration table automatically, and creates a read-only memory area at the desired address (default 0xFFFFF000). Since the configuration information is fixed, it can be efficiently implemented as a small ROM or with relatively few gates. A debug module (ahbreport) in the WORK.DEBUG package can be used to print the configuration table to the console during simulation, which is useful for debugging. A typical example is provided below:

```
VSIM 1> run
.
.
# LEON3 Actel PROASIC3-1000 Demonstration design
# GRLIB Version 1.0.16, build 2460
# Target technology: proasic3 ,  memory library: proasic3
# ahbctrl: AHB arbiter/multiplexer rev 1
# ahbctrl: Common I/O area disabled
# ahbctrl: AHB masters: 2, AHB slaves: 8
# ahbctrl: Configuration area at 0xfffff000, 4 kbyte
# ahbctrl: mst0: Gaisler Research        Leon3 SPARC V8 Processor
# ahbctrl: mst1: Gaisler Research        AHB Debug UART
# ahbctrl: slv0: European Space Agency   Leon2 Memory Controller
# ahbctrl:       memory at 0x00000000, size 512 Mbyte, cacheable, prefetch
# ahbctrl:       memory at 0x20000000, size 512 Mbyte
# ahbctrl:       memory at 0x40000000, size 1024 Mbyte, cacheable, prefetch
# ahbctrl: slv1: Gaisler Research        AHB/APB Bridge
# ahbctrl:       memory at 0x80000000, size 1 Mbyte
# ahbctrl: slv2: Gaisler Research        Leon3 Debug Support Unit
# ahbctrl:       memory at 0x90000000, size 256 Mbyte
# apbctrl: APB Bridge at 0x80000000 rev 1
# apbctrl: slv0: European Space Agency   Leon2 Memory Controller
# apbctrl:       I/O ports at 0x80000000, size 256 byte
# apbctrl: slv1: Gaisler Research        Generic UART
# apbctrl:       I/O ports at 0x80000100, size 256 byte
# apbctrl: slv2: Gaisler Research        Multi-processor Interrupt Ctrl.
# apbctrl:       I/O ports at 0x80000200, size 256 byte
# apbctrl: slv3: Gaisler Research        Modular Timer Unit
# apbctrl:       I/O ports at 0x80000300, size 256 byte
# apbctrl: slv7: Gaisler Research        AHB Debug UART
# apbctrl:       I/O ports at 0x80000700, size 256 byte
# apbctrl: slv11: Gaisler Research        General Purpose I/O port
# apbctrl:       I/O ports at 0x80000b00, size 256 byte
# grgpio11: 8-bit GPIO Unit rev 0
# gptimer3: GR Timer Unit rev 0, 8-bit scaler, 2 32-bit timers, irq 8
# irqmp: Multi-processor Interrupt Controller rev 3, #cpu 1
# apbuart1: Generic UART rev 1, fifo 1, irq 2
# ahbuart7: AHB Debug UART rev 0
# dsu3_2: LEON3 Debug support unit + AHB Trace Buffer, 1 kbytes
# leon3_0: LEON3 SPARC V8 processor rev 0
# leon3_0: icache 1*2 kbyte, dcache 1*2 kbyte
```

5.3.2 Device identification

The Identification Register contains three fields to identify uniquely an attached AHB unit: the vendor ID, the device ID, and the version number. The vendor ID is a unique number assigned to an IP vendor or organization. The device ID is a unique number assigned by a vendor to a specific IP core. The device ID is not related to the core's functionality. The version number can be used to identify (functionally) different versions of the unit.

The vendor IDs are declared in a package in each vendor library, usually called DEVICES. Vendor IDs are provided by Gaisler Research. The following ID's are currently assigned:

Vendor	ID
Gaisler Research	0x01
Pender Electronic Design	0x02
European Space Agency	0x04
Astrium EADS	0x06
OpenChip.org	0x07
OpenCores.org	0x08
Eonic BV	0x0B
Radionor	0x0F
Gleichmann Electronics	0x10
Menta	0x11
Sun Microsystems	0x13
Movidia	0x14
Orbita	0x17
Siemens AG	0x1A
Actel Corporation	0xAC
Caltech	0xCA
Embeddit	0xEA

TABLE 32. Vendor ID assignment

Vendor ID 0x00 is reserved to indicate that no core is present. Unused slots in the configuration table will have Identification Register set to 0.

5.3.3 Address decoding

The address mapping of AHB slaves in GRLIB is designed to be distributed, i.e. not rely on a shared static address decoder which must be modified as soon as a slave is added or removed. The GRLIB AHB bus controller, which implements the address decoder, will use the configuration information received from the slaves on HCONFIG to automatically generate the slave select signals (HSEL). When a slave is added or removed during the design, the address decoding function is automatically updated without requiring manual editing.

The AHB address range for each slave is defined by its Bank Address Registers (BAR). Address decoding is performed by comparing the 12-bit ADDR field in the BAR with part of the AHB address (HADDR). There are two types of banks defined for the AHB bus: AHB memory bank and AHB I/O bank. The AHB address decoding is done differently for the two types.

For AHB memory banks, the address decoding is performed by comparing the 12-bit ADDR field in the BAR with the 12 most significant bits in the AHB address (HADDR(31:20)). If equal, the corresponding HSEL will be generated. This means that the minimum address range occupied by an AHB memory bank is 1 MByte. To allow for larger address ranges, only the bits set in the MASK field of the BAR are compared. Consequently, HSEL will be generated when the following equation is true:

```
((BAR.ADDR xor HADDR[31:20]) and BAR.MASK) = 0
```

As an example, to decode a 16 MByte AHB memory bank at address 0x24000000, the ADDR field should be set to 0x240, and the MASK to 0xFF0.

For AHB I/O banks, the address decoding is performed by comparing the 12-bit ADDR field in the BAR with 12 bits in the AHB address (HADDR(19:8)). If equal, the corresponding HSEL will be generated. This means that the minimum address range occupied by an AHB I/O bank is 256 Byte. To allow for larger address ranges, only the bits set in the MASK field of the BAR are compared. Consequently, HSEL will be generated when the following equation is true:

```
((BAR.ADDR xor HADDR[19:8]) and BAR.MASK) = 0
```

The 12 most significant bits in the AHB address (HADDR(31:20)) are always fixed to 0xFFF, effectively placing all AHB I/O banks in the 0xFFF00000-0xFFFFEFFF address space. As an example, to decode an 4 kByte AHB I/O bank at address 0xFFF24000, the ADDR field should be set to 0x240, and the MASK to 0xFF0. Note: if HMASK = 0, the BAR is disabled rather than occupying the full AHB address range.

The AHB slaves in GRLIB define the value of their ADDR and MASK fields through generics. This allows to choose the address range for each slave when it is instantiated, without having to modify a central decoder or the slave itself. Below is an example of a component declaration of an AHB RAM memory, and how it can be instantiated:

```
component ahbram
  generic (
    hindex : integer := 0;                          -- AHB slave index
    haddr  : integer := 0;
    hmask  : integer := 16#fff#);
  port (
    rst    : in  std_ulogic;
    clk    : in  std_ulogic;
    hslvi  : in  ahb_slv_in_type;                   -- AHB slave input
    hslvo  : out ahb_slv_out_type);                   -- AHB slave output
end component;

ram0 : ahbram
  generic map (hindex => 1, haddr => 16#240#, hmask => 16#FF0#)
  port map (rst, clk, hslvi, hslvo(1));
```

An AHB slave can have up to four address mapping registers, thereby decode four independent areas in the AHB address space. HSEL is asserted when any of the areas is selected. To know which particular area was selected, the ahbsi record contains the additional bus signal HBSEL(0:3). The elements in HBSEL(0:3) are asserted if the corresponding to BAR(0-3) caused HSEL to be asserted. HBSEL is only valid when HSEL is asserted. For example, if BAR1 caused HSEL to be asserted, the HBSEL(1) will be asserted simultaneously with HSEL.

5.3.4 Cacheability

In processor-based systems without an MMU, the cacheable areas are typically defined statically in the cache controllers. The LEON3 processor builds the cachebility table automatically during synthesis, using the cacheability information in the AHB configuration records. In this way, the cacheability settings always reflect the current configuration.

For systems with an MMU, the cacheability information can be read out by from the configuration records through software. This allows the operating system to build an MMU page table with proper cacheable-bits set in the page table entries.

5.3.5 Interrupt steering

GRLIB provides a unified interrupt handling scheme by adding 32 interrupt signals (HIRQ) to the AHB bus, both as inputs and outputs. An AHB master or slave can drive as well as read any of the interrupts.

The output of each master includes all 32 interrupt signals in the vector ahbmo.hirq. An AHB master must therefore use a generic that specifies which HIRQ element to drive. This generic is of type integer, and typically called HIRQ (see example below).

```
component ahbmaster is
  generic (
    hindex : integer := 0;            -- master index
    hirq : integer := 0);           -- interrupt index
  port (
    reset   : in  std_ulogic;
    clk     : in  std_ulogic;
    hmsti   : in  ahb_mst_in_type;     -- AHB master inputs
    hmsto   : out ahb_mst_out_type     -- AHB master outputs
  );
end component;

master1 : ahbmaster
  generic map (hindex => 1, hirq => 1)
  port map (rst, clk, hmsti, hmsto(1));
```

The same applies to the output of each slave which includes all 32 interrupt signals in the vector ahbso.hirq. An AHB slave must therefore use a generic that specifies which HIRQ element to drive. This generic is of type integer, and typically called HIRQ (see example below).

```
component ahbslave
  generic (
    hindex : integer := 0;            -- slave index
    hirq : integer := 0);           -- interrupt index
  port (
    rst     : in  std_ulogic;
    clk     : in  std_ulogic;
    hslvi   : in  ahb_slv_in_type;     -- AHB slave inputs
    hslvo   : out ahb_slv_out_type);   -- AHB slave outputs
end component;

slave2 : ahbslave
  generic map (hindex => 2, hirq => 2)
  port map (rst, clk, hslvi, hslvo(1));
```

The AHB bus controller in the GRLIB provides interrupt combining. For each element in HIRQ, all the ahbmo.hirq signals from the AHB masters and all the ahbso.hirq signals from the AHB slaves are logically OR-ed. The combined result is output both on ahbmi.hirq (routed back to the AHB masters) and ahbsi.hirq (routed back to the AHB slaves). Consequently, the AHB masters and slaves share the same 32 interrupt signals.

An AHB unit that implements an interrupt controller can monitor the combined interrupt vector (either ahbsi.hirq or ahbmi.hirq) and generate the appropriate processor interrupt.

5.4 AMBA APB on-chip bus

5.4.1 General

The AMBA Advanced Peripheral Bus (APB) is a single-master bus suitable to interconnect units of low complexity which require only low data rates. An APB bus is interfaced with an AHB bus by means of a single AHB slave implementing the AHB/APB bridge. The AHB/APB bridge is the only APB master on one specific APB bus. More than one APB bus can be connected to one AHB bus, by means of multiple AHB/APB bridges. A conceptual view is provided in figure 8.

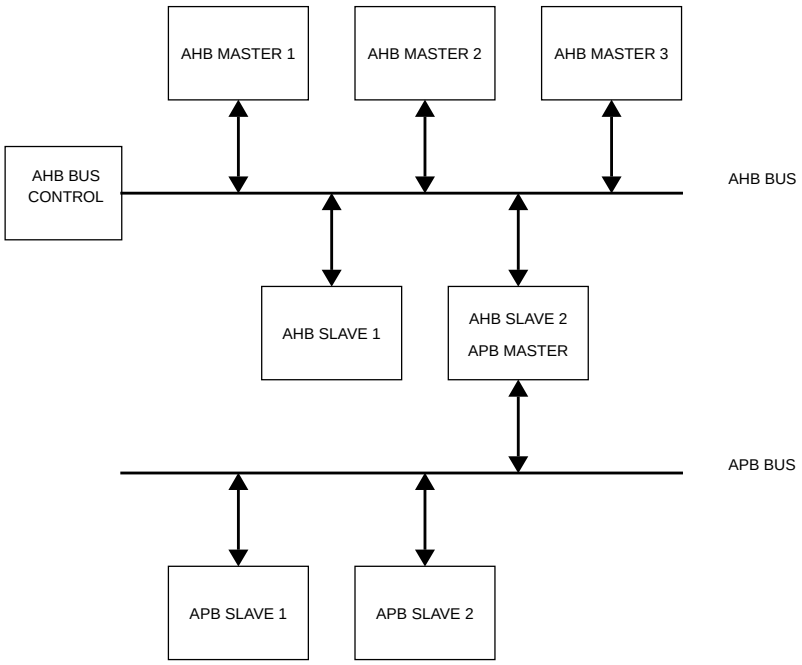


Figure 8. AMBA AHB/APB conceptual view

Since the APB bus is multiplexed (no tristate signals), a more correct view of the bus and the attached units can be seen in figure 9. The access to the AHB slave input (AHBI) is decoded and an access is made on APB bus. The APB master drives a set of signals grouped into a VHDL record called APBI which is sent to all APB slaves. The combined address decoder and bus multiplexer controls which slave is currently selected. The output record (APBO) of the active APB slave is selected by the bus multiplexer and forwarded to AHB slave output (AHBO).

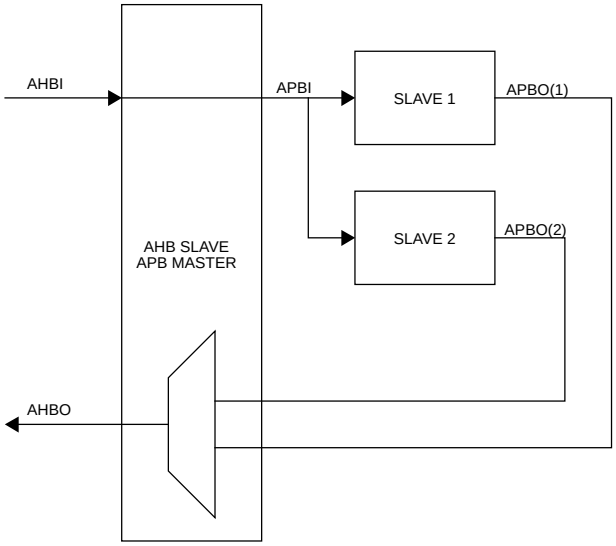


Figure 9. APB inter-connection view

5.4.2 APB slave interface

The APB slave inputs and outputs are defined as VHDL record types, and are exported through the TYPES package in the GRLIB AMBA library:

```
-- APB slave inputs
  type apb_slv_in_type is record
    psel    : std_logic_vector(0 to NAPBSLV-1);       -- slave select
    penable : std_ulogic;                             -- strobe
    paddr   : std_logic_vector(31 downto 0);          -- address bus (byte)
    pwrite  : std_ulogic;                             -- write
    pwdata  : std_logic_vector(31 downto 0);          -- write data bus
    pirq    : std_logic_vector(NAHBIRQ-1 downto 0);   -- interrupt result bus
  end record;

-- APB slave outputs
  type apb_slv_out_type is record
    prdata   : std_logic_vector(31 downto 0);           -- read data bus
    pirq     : std_logic_vector(NAHBIRQ-1 downto 0);    -- interrupt bus
    pconfig  : apb_config_type;                         -- memory access reg.
    pindex   : integer range 0 to NAPBSLV -1;           -- diag use only
  end record;
```

The elements in the record types correspond to the APB signals as defined in the AMBA 2.0 specification, with the addition of three sideband signals: PCONFIG, PIRQ and PINDEX. A typical APB slave in GRLIB has the following definition:

```
library grlib;
use grlib.amba.all;
library ieee;
use ieee.std_logic.all;

entity apbslave is
  generic (
    pindex  : integer := 0);           -- slave bus index
  port (
    rst     : in  std_ulogic;
    clk     : in  std_ulogic;
    apbi    : in  apb_slv_in_type;     -- APB slave inputs
    apbo    : out apb_slv_out_type     -- APB slave outputs
  );
end entity;
```

The input record (APBI) is routed to all slaves, and include the select signals for all slaves in the vector APBI.PSEL. An APB slave must therefore use a generic that specifies which PSEL element to use. This generic is of type integer, and typically called PINDEX (see example above).

5.4.3 AHB/APB bridge

GRLIB provides a combined AHB slave, APB bus master, address decoder and bus multiplexer. It receives the AHBI and AHBO records from the AHB bus, and generates APBI and APBO records on the APB bus. The address decoding function will drive one of the APBI.PSEL elements to indicate the selected APB slave. The bus multiplexer function will select from which APB slave data will be taken to drive the AHBI signal. A typical APB master in GRLIB has the following definition:

```
library IEEE;
use IEEE.std_logic_1164.all;
library grlib;
use grlib.amba.all;

entity apbmst is
  generic (
    hindex  : integer := 0;            -- AHB slave bus index
  );
  port (
    rst     : in  std_ulogic;
    clk     : in  std_ulogic;
    ahbi    : in  ahb_slv_in_type;     -- AHB slave inputs
    ahbo    : out ahb_slv_out_type;    -- AHB slave outputs
    apbi    : out apb_slv_in_type;     -- APB master inputs
    apbo    : in  apb_slv_out_vector   -- APB master outputs
  );
end;
```

5.4.4 APB bus index control

The APB slave output records contain the sideband signal PINDEX. This signal is used to verify that the slave is driving the correct element of the AHBPO bus. The generic PINDEX that is used to select the appropriate PSEL is driven back on APBO.PINDEX. The APB controller then checks that the value of the received PINDEX is equal to the bus index. An error is issued during simulation if a mismatch is detected.

5.5 APB plug&play configuration

5.5.1 General

The GRLIB implementation of the APB bus includes the same type of mechanism to provide plug&play support as for the AHB bus. The plug&play support consists of three parts: identification of attached slaves, address mapping, and interrupt routing. The plug&play information for each APB slave consists of a configuration record containing two 32-bit words. The first word is called the identification register and contains information on the device type and interrupt routing. The last word is the bank address register (BAR) and contains address mapping information for the APB slave. Only a single BAR is defined per APB slave. An APB slave is neither prefetchable nor cacheable.

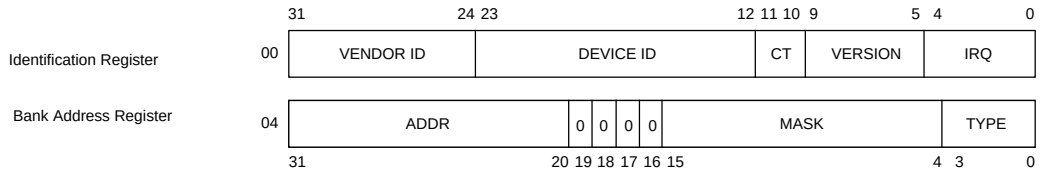


Figure 10. APB plug&play configuration layout

All addressing of the APB is referenced to the AHB address space. The 12 most significant bits of the AHB bus address are used for addressing the AHB slave of the AHB/APB bridge, leaving the 20 least significant bits for APB slave addressing.

The plug&play information for all attached APB slaves appear as a read-only table mapped on a fixed address of the AHB, typically at 0x---FF000. The configuration records of the APB slaves appear in 0x---FF000 - 0x---FFF00 on the AHB bus. Since each record is 2 words (8 bytes), the table has space for 512 slaves on a signle APB bus. A plug&play operating system (or any other application) can scan the configuration table and automatically detect which units are present on the APB bus, how they are configured, and where they are located (slaves).

The configuration record from each APB unit is sent to the APB bus controller via the PCONFIG signal. The bus controller creates the configuration table automatically, and creates a read-only memory area at the desired address (default 0x---FF000). Since the configuration information is fixed, it can be efficiently implemented as a small ROM or with relatively few gates. A special reporting module (apbreport) is provided in the WORK.DEBUG package of Grlib which can be used to print the configuration table to the console during simulation.

5.5.2 Device identification

The APB bus uses same type of Identification Register as previously defined for the AHB bus.

5.5.3 Address decoding

The address mapping of APB slaves in GRLIB is designed to be distributed, i.e. not rely on a shared static address decoder which must be modified as soon as a slave is added or removed. The GRLIB APB master, which implements the address decoder, will use the configuration information received from the slaves on PCONFIG to automatically generate the slave select signals (PSEL). When a slave is added or removed during the design, the address decoding function is automatically updated without requiring manual editing.

The APB address range for each slave is defined by its Bank Address Registers (BAR). There is one type of banks defined for the APB bus: APB I/O bank. Address decoding is performed by comparing the 12-bit ADDR field in the BAR with 12 bits in the AHB address (HADDR(19:8)). If equal, the corresponding PSEL will be generated. This means that the minimum address range occupied by an APB I/O bank is 256 Byte. To allow for larger address ranges, only the bits set in the MASK field of the BAR are compared. Consequently, PSEL will be generated when the following equation is true:

```
((BAR.ADDR xor HADDR[19:8]) and BAR.MASK) = 0
```

As an example, to decode an 4 kByte AHB I/O bank at address 0x---24000, the ADDR field should be set to 0x240, and the MASK to 0xFF0. Note that the 12 most significant bits of AHBI.HADDR are used for addressing the AHB slave of the AHB/APB bridge, leaving the 20 least significant bits for APB slave addressing.

As for AHB slaves, the APB slaves in GRLIB define the value of their ADDR and MASK fields through generics. This allows to choose the address range for each slave when it is instantiated, without having to modify a central decoder or the slave itself. Below is an example of a component declaration of an APB I/O unit, and how it can be instantiated:

```
component apbio
  generic (
    pindex : integer := 0;
    paddr  : integer := 0;
    pmask  : integer := 16#fff#);
  port (
    rst    : in  std_ulogic;
    clk    : in  std_ulogic;
    apbi   : in  apb_slv_in_type;
    apbo   : out apb_slv_out_type);
end component;

io0 : apbio
  generic map (pindex => 1, paddr => 16#240#, pmask => 16#FF0#)
  port map (rst, clk, apbi, apbo(1));
```

5.5.4 Interrupt steering

GRLIB provides a unified interrupt handling scheme by also adding 32 interrupt signals (PIRQ) to the APB bus, both as inputs and outputs. An APB slave can drive as well as read any of the interrupts. The output of each slave includes all 32 interrupt signals in the vector APBO.PIRQ. An APB slave must therefore use a generic that specifies which PIRQ element to drive. This generic is of type integer, and typically called PIRQ (see example below).

```
component apbslave
  generic (
    pindex : integer := 0;            -- slave index
    pirq   : integer := 0);           -- interrupt index
  port (
    rst     : in  std_ulogic;
    clk     : in  std_ulogic;
    apbi    : in  apb_slv_in_type;     -- APB slave inputs
    apbo    : out apb_slv_out_type);   -- APB slave outputs
end component;

slave3 : apbslave
  generic map (pindex => 1, pirq => 2)
  port map (rst, clk, pslvi, pslvo(1));
```

The AHB/APB bridge in the GRLIB provides interrupt combining, and merges the APB-generated interrupts with the interrups bus on the AHB bus. This is done by OR-ing the 32-bit interrupt vectors from each APB slave into one joined vector, and driving the combined value on the AHB slave output bus (AHBSO.HIRQ). The APB interrupts will then be merged with the AHB interrupts. The resulting interrupt vector in available on the AHB slave input (AHBSI.HIRQ), and is also driven on the APB slave inputs (APBI.PIRQ) by the AHB/APB bridge. Each APB slave (as well as AHB slave) thus sees the combined AHB/APB interrupts. An interrupt controller can then be placed either on the AHB or APB bus and still monitor all interrupts.

5.6 Technology mapping

5.6.1 General

GRLIB provides portability support for both ASIC and FPGA technologies. The support is implemented by means of encapsulation of technology specific components such as memories, pads and clock buffers. The interface to the encapsulated component is made technology independent, not relying on any specific VHDL or Verilog code provided by the foundry or FPGA manufacturer. The interface to the component stays therefore always the same. No modification of the design is therefore required if a different technology is targeted. The following technologies are currently supported by the TECHMAP.GENCOMP package:

```
constant inferred    : integer := 0;
constant virtex      : integer := 1;
constant virtex2     : integer := 2;
constant memvirage   : integer := 3;
constant axcel       : integer := 4;
constant proasic     : integer := 5;
constant atc18s      : integer := 6;
constant altera      : integer := 7;
constant umc         : integer := 8;
constant rhumc       : integer := 9;
constant apa3        : integer := 10;
constant spartan3    : integer := 11;
constant ihp25       : integer := 12;
constant rhlib18t    : integer := 13;
constant virtex4     : integer := 14;
constant lattice     : integer := 15;
constant ut25        : integer := 16;
constant spartan3e   : integer := 17;
constant peregrine   : integer := 18;
constant memartisan  : integer := 19;
constant virtex5     : integer := 20;
constant custom1     : integer := 21;
constant ihp25rh     : integer := 22;
constant stratix1    : integer := 23;
constant stratix2    : integer := 24;
constant eclipse     : integer := 25;
```

```
constant stratix3    : integer := 26;
constant cyclone3    : integer := 27;
constant memvirage90 : integer := 28;
constant tsmc90      : integer := 29;
```

Each encapsulating component provides a VHDL generic (normally named TECH) with which the targeted technology can be selected. The generic is used by the component to select the correct technology specific cells to instantiatein its architecture and to configure them approriately. This method does not rely on the synthesis tool to inferring the correct cells.

For technologies not defined in GRLIB, the default “inferred” option can be used. This option relies on the synthesis tool to infer the correct technology cells for the targeted device.

A second VHDL generic (normally named MEMTECH) is used for selecting the memory cell technology. This is useful for ASIC technologies where the pads are provided by the foundry and the memory cells are provided by a different source. For memory cells, generics are also used to specify the address and data widths, and the number of ports.

The two generics TECH and MEMTECH should be defined at the top level entity of a design and be propagated to all underlying components supporting technology specific implementations.

5.6.2 Memory blocks

Memory blocks are often implemented with technology specific cells or macrocells and require an encapsulating component to offer a unified technology independent interface. The TECHMAP library provides such technology independent memory component, as the synchronous single-port RAM shown in the following code example. The address and data widths are fully configurable by means of the generics ABITS and DBITS, respectively.

```
component syncram
  generic (
    memtech  : integer := 0;                          -- memory technology
    abits    : integer := 6;                          -- address width
    dbits    : integer := 8);                         -- data width
  port (
    clk      : in  std_ulogic;
    address  : in  std_logic_vector((abits -1) downto 0);
    datain   : in  std_logic_vector((dbits -1) downto 0);
    dataout  : out std_logic_vector((dbits -1) downto 0);
    enable   : in  std_ulogic;
    write    : in  std_ulogic);
end component;
```

This synchronous single-port RAM component is used in the AHB RAM component shown in the following code example.

```
component ahbram
  generic (
    hindex : integer := 0;                          -- AHB slave index
    haddr  : integer := 0;
    hmask  : integer := 16#fff#;
    memtech : integer := 0;                          -- memory technology
    kbytes  : integer := 1);                         -- memory size
  port (
    rst     : in  std_ulogic;
    clk     : in  std_ulogic;
    hslvi   : in  ahb_slv_in_type;                   -- AHB slave input
    hslvo   : out ahb_slv_out_type);                 -- AHB slave output
end component;

ram0 : ahbram
  generic map (hindex => 1, haddr => 16#240#, hmask => 16#FF0#,
               tech => virtex, kbytes => 4)
  port map (rst, clk, hslvi, hslvo(1));
```

In addition to the selection of technology (VIRTEX in this case), the size of the AHB RAM is specified in number of kilo-bytes. The conversion from kilo-bytes to the number of address bits is performed automatically in the AHB RAM component. In this example, the data width is fixed to 32 bits and requires no generic. The VIRTEX constant used in this example is defined in the TECHMAP.GENCOMP package.

5.6.3 Pads

As for memory cells, the pads used in a design are always technology dependent. The TECHMAP library provides a set of encapsulated components that hide all the technology specific details from the user. In addition to the VHDL generic used for selecting the technology (normally named TECH), generics are provided for specifying the input/output technology levels, voltage levels, slew and driving strength. A typical open-drain output pad is shown in the following code example:

```
component odpad
  generic (
    tech     : integer := 0;
    level    : integer := 0;
    slew     : integer := 0;
    voltage  : integer := 0;
    strength : integer := 0
  );
  port (
    pad      : out std_ulogic;
    o        : in  std_ulogic
  );
end component;

pad0 : odpad
  generic map (tech => virtex, level => pci33, voltage => x33v)
  port map (pad => pci_irq, o => irqn);
```

The TECHMAP.GENCOMP package defines the following constants that to be used for configuring pads:

```
-- input/output voltage

constant x18v      : integer := 1;
constant x25v      : integer := 2;
constant x33v      : integer := 3;
constant x50v      : integer := 5;

-- input/output levels

constant ttl      : integer := 0;
constant cmos     : integer := 1;
constant pci33    : integer := 2;
constant pci66    : integer := 3;
constant lvds     : integer := 4;
constant sstl2_i  : integer := 5;
constant sstl2_ii : integer := 6;
constant sstl3_i  : integer := 7;
constant sstl3_ii : integer := 8;

-- pad types

constant normal   : integer := 0;
constant pullup   : integer := 1;
constant pulldown : integer := 2;
constant opendrain: integer := 3;
constant schmitt  : integer := 4;
constant dci      : integer := 5;
```

The slew control and driving strength is not supported by all target technologies, or is often implemented differently between different technologie. The documentation for the IP core implementing the pad should be consulted for details.

5.7 Scan test support

To support scan test methods, the GRLIB AHB and APB bus records include four extra signals: testen (test enable), scanen (scan enable), testoen (bidir control) and testrst (test reset). Scan methodology requires that all flip-flops are controllable in test mode, i.e. that they are connected to the same clock and that asynchronous resets are connected to the test reset signal. Bi-directionla or tri-state outputs should also be controllable. The four test signals are driven from the AHB bus controller (ahbctrl), where they are defined as optional inputs. The test signals are the routed to the inputs of all AHB masters and slaves (ahbmi and ahbsi records). The APB master (apbctrl) routes the test signals further to all APB slaves using the apbi record. In this way, the scan test control signals are available in all AMBA cores without additional external connections.

Cores which use the scan signals include LEON3, MCTRL and GRGPIO.

Beilage 3

LEON3 GRLIB Auszug

55 LEON3 - High-performance SPARC V8 32-bit Processor

55.1 Overview

LEON3 is a 32-bit processor core conforming to the IEEE-1754 (SPARC V8) architecture. It is designed for embedded applications, combining high performance with low complexity and low power consumption.

The LEON3 core has the following main features: 7-stage pipeline with Harvard architecture, separate instruction and data caches, hardware multiplier and divider, on-chip debug support and multiprocessor extensions.

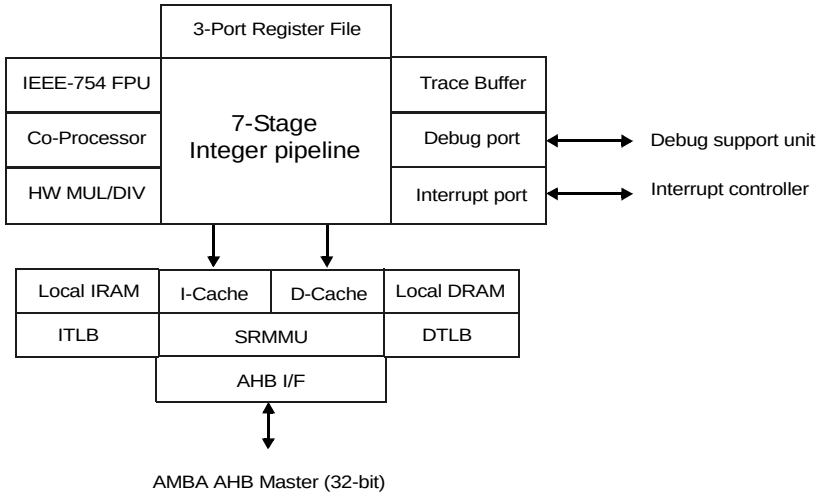


Figure 173. LEON3 processor core block diagram

Note: this manual describes the full functionality of the LEON3 core. Through the use of VHDL generics, parts of the described functionality can be suppressed or modified to generate a smaller or faster implementation.

55.1.1 Integer unit

The LEON3 integer unit implements the full SPARC V8 standard, including hardware multiply and divide instructions. The number of register windows is configurable within the limit of the SPARC standard (2 - 32), with a default setting of 8. The pipeline consists of 7 stages with a separate instruction and data cache interface (Harvard architecture).

55.1.2 Cache sub-system

LEON3 has a highly configurable cache system, consisting of a separate instruction and data cache. Both caches can be configured with 1 - 4 sets, 1 - 256 kbyte/set, 16 or 32 bytes per line. Sub-blocking is implemented with one valid bit per 32-bit word. The instruction cache uses streaming during line-refill to minimize refill latency. The data cache uses write-through policy and implements a double-word write-buffer. The data cache can also perform bus-snooping on the AHB bus. A local scratch pad ram can be added to both the instruction and data cache controllers to allow 0-waitstates access memory without data write back.

55.1.3 Floating-point unit and co-processor

The LEON3 integer unit provides interfaces for a floating-point unit (FPU), and a custom co-processor. Two FPU controllers are available, one for the high-performance GRFPU (available from Gaisler Research) and one for the Meiko FPU core (available from Sun Microsystems). The floating-point processors and co-processor execute in parallel with the integer unit, and does not block the operation unless a data or resource dependency exists.

55.1.4 Memory management unit

A SPARC V8 Reference Memory Management Unit (SRMMU) can optionally be enabled. The SRMMU implements the full SPARC V8 MMU specification, and provides mapping between multiple 32-bit virtual address spaces and 36-bit physical memory. A three-level hardware table-walk is implemented, and the MMU can be configured to up to 64 fully associative TLB entries.

55.1.5 On-chip debug support

The LEON3 pipeline includes functionality to allow non-intrusive debugging on target hardware. To aid software debugging, up to four watchpoint registers can be enabled. Each register can cause a breakpoint trap on an arbitrary instruction or data address range. When the (optional) debug support unit is attached, the watchpoints can be used to enter debug mode. Through a debug support interface, full access to all processor registers and caches is provided. The debug interfaces also allows single stepping, instruction tracing and hardware breakpoint/watchpoint control. An internal trace buffer can monitor and store executed instructions, which can later be read out over the debug interface.

55.1.6 Interrupt interface

LEON3 supports the SPARC V8 interrupt model with a total of 15 asynchronous interrupts. The interrupt interface provides functionality to both generate and acknowledge interrupts.

55.1.7 AMBA interface

The cache system implements an AMBA AHB master to load and store data to/from the caches. The interface is compliant with the AMBA-2.0 standard. During line refill, incremental burst are generated to optimise the data transfer.

55.1.8 Power-down mode

The LEON3 processor core implements a power-down mode, which halts the pipeline and caches until the next interrupt. This is an efficient way to minimize power-consumption when the application is idle, and does not require tool-specific support in form of clock gating. To implement clock-gating, a suitable clock-enable signal is produced by the processor.

55.1.9 Multi-processor support

LEON3 is designed to be use in multi-processor systems. Each processor has a unique index to allow processor enumeration. The write-through caches and snooping mechanism guarantees memory coherency in shared-memory systems.

55.1.10 Performance

Using 8K + 8K caches and a 16x16 multiplier, the dhrystone 2.1 benchmark reports 1,500 iteration/s/MHz using the gcc-3.4.4 compiler (-O2). This translates to 0.85 dhrystone MIPS/MHz using the VAX 11/780 value a reference for one MIPS.

55.2 LEON3 integer unit

55.2.1 Overview

The LEON3 integer unit implements the integer part of the SPARC V8 instruction set. The implementation is focused on high performance and low complexity. The LEON3 integer unit has the following main features:

- 7-stage instruction pipeline
- Separate instruction and data cache interface
- Support for 2 - 32 register windows
- Hardware multiplier with optional 16x16 bit MAC and 40-bit accumulator
- Radix-2 divider (non-restoring)
- Single-vector trapping for reduced code size

Figure 174 shows a block diagram of the integer unit.

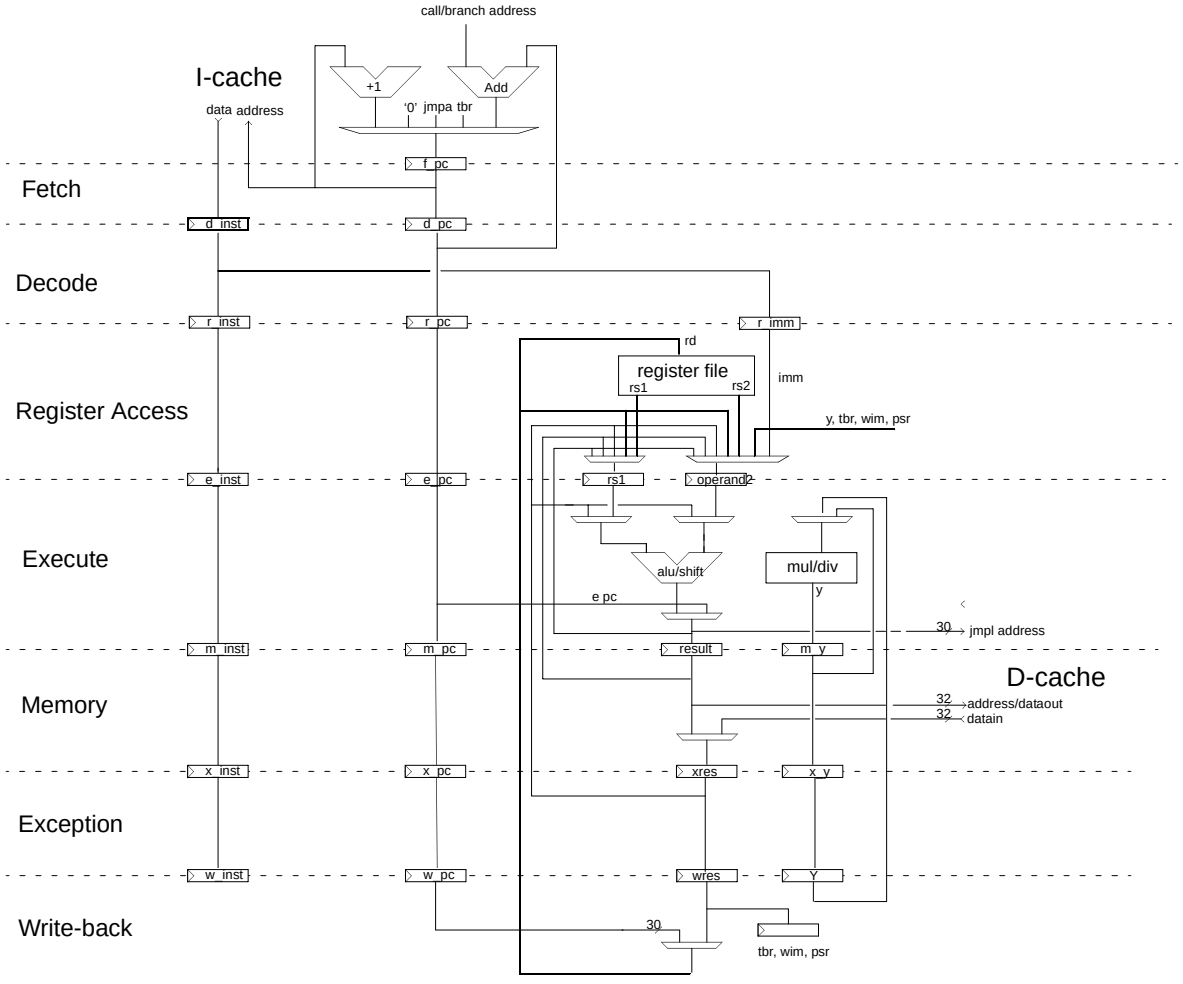


Figure 174. LEON3 integer unit datapath diagram

55.2.2 Instruction pipeline

The LEON integer unit uses a single instruction issue pipeline with 7 stages:

1. FE (Instruction Fetch): If the instruction cache is enabled, the instruction is fetched from the instruction cache. Otherwise, the fetch is forwarded to the memory controller. The instruction is valid at the end of this stage and is latched inside the IU.

2. DE (Decode): The instruction is decoded and the CALL and Branch target addresses are generated.

3. RA (Register access): Operands are read from the register file or from internal data bypasses.

4. EX (Execute): ALU, logical, and shift operations are performed. For memory operations (e.g., LD) and for JMPL/RETT, the address is generated.

5. ME (Memory): Data cache is accessed. Store data read out in the execution stage is written to the data cache at this time.

6. XC (Exception) Traps and interrupts are resolved. For cache reads, the data is aligned as appropriate.

7. WR (Write): The result of any ALU, logical, shift, or cache operations are written back to the register file.

Table 623 lists the cycles per instruction (assuming cache hit and no icc or load interlock):

*Table 623.*Instruction timing

Instruction	**Cycles (MMU disabled)**	**Cycles (MMU fast-write)**	**Cycles (MMU slow-write)**
JMPL, RETT	3	3	3
Double load	2	2	2
Single store	2	2	4
Double store	3	3	5
SMUL/UMUL	1/4*	1/4*	1/4*
SDIV/UDIV	35	35	35
Taken Trap	5	5	5
Atomic load/store	3	3	5
All other instructions	**1**	**1**	**1**

* Multiplication cycle count is 1 clock for the 32x32 multiplier and 4 clocks for the 16x16 version.

55.2.3 SPARC Implementor's ID

Gaisler Research is assigned number 15 (0xF) as SPARC implementor's identification. This value is hard-coded into bits 31:28 in the %psr register. The version number for LEON3 is 3, which is hard-coded in to bits 27:24 of the %psr.

55.2.4 Divide instructions

Full support for SPARC V8 divide instructions is provided (SDIV, UDIV, SDIVCC & UDIVCC). The divide instructions perform a 64-by-32 bit divide and produce a 32-bit result. Rounding and overflow detection is performed as defined in the SPARC V8 standard.

55.2.5 Multiply instructions

The LEON processor supports the SPARC integer multiply instructions UMUL, SMUL UMULCC and SMULCC. These instructions perform a 32x32-bit integer multiply, producing a 64-bit result. SMUL and SMULCC performs signed multiply while UMUL and UMULCC performs unsigned multiply. UMULCC and SMULCC also set the condition codes to reflect the result. The multiply instructions are performed using a 32x32 pipelined hardware multiplier, or a 16x16 hardware multiplier which is iterated four times. To improve the timing, the 16x16 multiplier can optionally be provided with a pipeline stage.

55.2.6 Multiply and accumulate instructions

To accelerate DSP algorithms, two multiply&accumulate instructions are implemented: UMAC and SMAC. The UMAC performs an unsigned 16-bit multiply, producing a 32-bit result, and adds the result to a 40-bit accumulator made up by the 8 lsb bits from the %y register and the %asr18 register. The least significant 32 bits are also written to the destination register. SMAC works similarly but performs signed multiply and accumulate. The MAC instructions execute in one clock but have two clocks latency, meaning that one pipeline stall cycle will be inserted if the following instruction uses the destination register of the MAC as a source operand.

Assembler syntax:

```
umacrs1, reg_imm, rd
smacrs1, reg_imm, rd
```

Operation:

```
prod[31:0] = rs1[15:0] * reg_imm[15:0]
result[39:0] = (Y[7:0] & %asr18[31:0]) + prod[31:0]
(Y[7:0] & %asr18[31:0]) = result[39:0]
rd = result[31:0]
```

%asr18 can be read and written using the RDASR and WRASR instructions.

55.2.7 Hardware breakpoints

The integer unit can be configured to include up to four hardware breakpoints. Each breakpoint consists of a pair of application-specific registers (%asr24/25, %asr26/27, %asr28/29 and %asr30/31) registers; one with the break address and one with a mask:

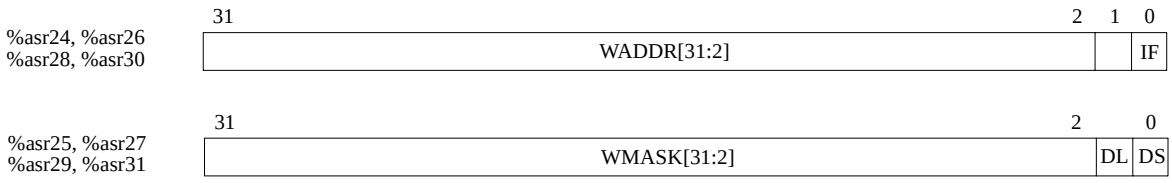


Figure 175. Watch-point registers

Any binary aligned address range can be watched - the range is defined by the WADDR field, masked by the WMASK field (WMASK[x] = 1 enables comparison). On a breakpoint hit, trap 0x0B is generated. By setting the IF, DL and DS bits, a hit can be generated on instruction fetch, data load or data store. Clearing these three bits will effectively disable the breakpoint function.

55.2.8 Instruction trace buffer

The instruction trace buffer consists of a circular buffer that stores executed instructions. The trace buffer operation is controlled through the debug support interface, and does not affect processor operation (see the DSU description). The size of the trace buffer is configurable from 1 to 64 kB through a VHDL generic. The trace buffer is 128 bits wide, and stores the following information:

- Instruction address and opcode
- Instruction result
- Load/store data and address
- Trap information
- 30-bit time tag

The operation and control of the trace buffer is further described in section 29.4. Note that in multi-processor systems, each processor has its own trace buffer allowing simultaneous tracing of all instruction streams.

55.2.9 Processor configuration register

The application specific register 17 (%asr17) provides information on how various configuration options were set during synthesis. This can be used to enhance the performance of software, or to support enumeration in multi-processor systems. The register can be accessed through the RDASR instruction, and has the following layout:

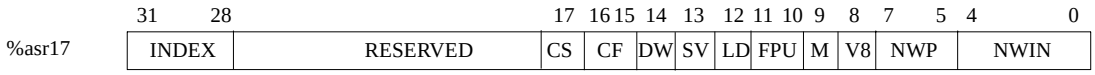


Figure 176. LEON3 configuration register (%asr17)

Field Definitions:

[31:28]: Processor index. In multi-processor systems, each LEON core gets a unique index to support enumeration. The value in this field is identical to the *hindex* generic parameter in the VHDL model.

value in this field is identical to the *hindex* generic parameter in the VHDL model.

[17]: Clock switching enabled (CS). If set switching between AHB and CPU frequency is available.

[16:15]: CPU clock frequency (CF). CPU core runs at (CF+1) times AHB frequency.

[14]: Disable write error trap (DWT). When set, a write error trap (tt = 0x2b) will be ignored. Set to zero after reset.

[13]: Single-vector trapping (SVT) enable. If set, will enable single-vector trapping. Fixed to zero if SVT is not implemented. Set to zero after reset.

[12]: Load delay. If set, the pipeline uses a 2-cycle load delay. Otherwise, a 1-cycle load delay i s used. Generated from the *lddel* generic parameter in the VHDL model.

[11:10]: FPU option. “00” = no FPU; “01” = GRFPU; “10” = Meiko FPU, “11” = GRFPU-Lite

[9]: If set, the optional multiply-accumulate (MAC) instruction is available

[8]: If set, the SPARC V8 multiply and divide instructions are available.

[7:5]: Number of implemented watchpoints (0 - 4)

[4:0]: Number of implemented registers windows corresponds to NWIN+1.

55.2.10 Exceptions

LEON adheres to the general SPARC trap model. The table below shows the implemented traps and their individual priority. When PSR (processor status register) bit ET=0, an exception trap causes the processor to halt execution and enter error mode, and the external error signal will then be asserted.

*Table 624.*Trap allocation and priority

Trap	TT	Pri	Description
reset	0x00	1	Power-on reset
write error	0x2b	2	write buffer error
instruction_access_error	0x01	3	Error during instruction fetch
illegal_instruction	0x02	5	UNIMP or other un-implemented instruction
privileged_instruction	0x03	4	Execution of privileged instruction in user mode
fp_disabled	0x04	6	FP instruction while FPU disabled
cp_disabled	0x24	6	CP instruction while Co-processor disabled
watchpoint_detected	0x0B	7	Hardware breakpoint match
window_overflow	0x05	8	SAVE into invalid window
window_underflow	0x06	8	RESTORE into invalid window
register_hadrware_error	0x20	9	register file EDAC error (LEON-FT only)
mem_address_not_aligned	0x07	10	Memory access to un-aligned address
fp_exception	0x08	11	FPU exception
cp_exception	0x28	11	Co-processor exception
data_access_exception	0x09	13	Access error during load or store instruction
tag_overflow	0x0A	14	Tagged arithmetic overflow
divide_exception	0x2A	15	Divide by zero
interrupt_level_1	0x11	31	Asynchronous interrupt 1
interrupt_level_2	0x12	30	Asynchronous interrupt 2
interrupt_level_3	0x13	29	Asynchronous interrupt 3
interrupt_level_4	0x14	28	Asynchronous interrupt 4
interrupt_level_5	0x15	27	Asynchronous interrupt 5
interrupt_level_6	0x16	26	Asynchronous interrupt 6
interrupt_level_7	0x17	25	Asynchronous interrupt 7
interrupt_level_8	0x18	24	Asynchronous interrupt 8
interrupt_level_9	0x19	23	Asynchronous interrupt 9
interrupt_level_10	0x1A	22	Asynchronous interrupt 10
interrupt_level_11	0x1B	21	Asynchronous interrupt 11
interrupt_level_12	0x1C	20	Asynchronous interrupt 12
interrupt_level_13	0x1D	19	Asynchronous interrupt 13
interrupt_level_14	0x1E	18	Asynchronous interrupt 14
interrupt_level_15	0x1F	17	Asynchronous interrupt 15
trap_instruction	0x80 - 0xFF	16	Software trap instruction (TA)

55.2.11 Single vector trapping (SVT)

Single-vector trapping (SVT) is an SPARC V8e option to reduce code size for embedded applications. When enabled, any taken trap will always jump to the reset trap handler (%tbr.tba + 0). The trap type

will be indicated in %tbr.tt, and must be decoded by the shared trap handler. SVT is enabled by setting bit 13 in %asr17. The model must also be configured with the SVT generic = 1.

55.2.12 Address space identifiers (ASI)

In addition to the address, a SPARC processor also generates an 8-bit address space identifier (ASI), providing up to 256 separate, 32-bit address spaces. During normal operation, the LEON3 processor accesses instructions and data using ASI 0x8 - 0xB as defined in the SPARC standard. Using the LDA/STA instructions, alternative address spaces can be accessed. The table shows the ASI usage for LEON. Only ASI[5:0] are used for the mapping, ASI[7:6] have no influence on operation.

*Table 625.*ASI usage

ASI	Usage
0x01	Forced cache miss
0x02	System control registers (cache control register)
0x08, 0x09, 0x0A, 0x0B	Normal cached access (replace if cacheable)
0x0C	Instruction cache tags
0x0D	Instruction cache data
0x0E	Data cache tags
0x0F	Data cache data
0x10	Flush instruction cache
0x11	Flush data cache

55.2.13 Power-down

The processor can be configured to include a power-down feature to minimize power consumption during idle periods. The power-down mode is entered by performing a WRASR instruction to %asr19:

```
wr %g0, %asr19
```

During power-down, the pipeline is halted until the next interrupt occurs. Signals inside the processor pipeline and caches are then static, reducing power consumption from dynamic switching.

55.2.14 Processor reset operation

The processor is reset by asserting the RESET input for at least 4 clock cycles. The following table indicates the reset values of the registers which are affected by the reset. All other registers maintain their value (or are undefined).

*Table 626.*Processor reset values

Register	Reset value
PC (program counter)	0x0
nPC (next program counter)	0x4
PSR (processor status register)	ET=0, S=1

By default, the execution will start from address 0. This can be overridden by setting the RSTADDR generic in the model to a non-zero value. The reset address is always aligned on a 4 kbyte boundary. If RSTADDR is set to 16#FFFFF#, then the reset address is taken from the signal IRQI.RSTVEC. This allows the reset address to be changed dynamically.

55.2.15 Multi-processor support

The LEON3 processor support synchronous multi-processing (SMP) configurations, with up to 16 processors attached to the same AHB bus. In multi-processor systems, only the first processor will start. All other processors will remain halted in power-down mode. After the system has been initial-

ized, the remaining processors can be started by writing to the ‘MP status register’, located in the multi-processor interrupt controller. The halted processors start executing from the reset address (0 or RSTADDR generic). Enabling SMP is done by setting the *smp* generic to 1 or higher. Cache snooping should always be enabled in SMP systems to maintain data cache coherency between the processors.

55.2.16 Cache sub-system

The LEON3 processor implements a Harvard architecture with separate instruction and data buses, connected to two independent cache controllers. Both instruction and data cache controllers can be separately configured to implement a direct-mapped cache or a multi-set cache with set associativity of 2 - 4. The set size is configurable to 1 - 256 kbyte, divided into cache lines with 16 or 32 bytes of data. In multi-set configurations, one of three replacement policies can be selected: least-recently-used (LRU), least-recently-replaced (LRR) or (pseudo-) random. If the LRR algorithm can only be used when the cache is 2-way associative. A cache line can be locked in the instruction or data cache preventing it from being replaced by the replacement algorithm.

NOTE: The LRR algorithm uses one extra bit in tag rams to store replacement history. The LRU algorithm needs extra flip-flops per cache line to store access history. The random replacement algorithm is implemented through modulo-N counter that selects which line to evict on cache miss.

Cachability for both caches is controlled through the AHB plug&play address information. The memory mapping for each AHB slave indicates whether the area is cachable, and this information is used to (statically) determine which access will be treated as cacheable. This approach means that the cachability mapping is always coherent with the current AHB configuration. The AMBA plug&play cachability can be overriden using the CACHED generic. When this generic is not zero, it is treated as a 16-bit field, defining the cachability of each 256 Mbyte address block on the AMBA bus. A value of 16#00F3# will thus define cachable areas in 0 - 0x20000000 and 0x40000000 - 0x80000000..

55.3 Instruction cache

55.3.1 Operation

The instruction cache can be configured as a direct-mapped cache or as a multi-set cache with associativity of 2 - 4 implementing either LRU or random replacement policy or as 2-way associative cache implementing LRR algorithm. The set size is configurable to 1 - 64 kbyte and divided into cache lines of 16- 32 bytes. Each line has a cache tag associated with it consisting of a tag field, valid field with one valid bit for each 4-byte sub-block and optional LRR and lock bits. On an instruction cache miss to a cachable location, the instruction is fetched and the corresponding tag and data line updated. In a multi-set configuration a line to be replaced is chosen according to the replacement policy.

If instruction burst fetch is enabled in the cache control register (CCR) the cache line is filled from main memory starting at the missed address and until the end of the line. At the same time, the instructions are forwarded to the IU (streaming). If the IU cannot accept the streamed instructions due to internal dependencies or multi-cycle instruction, the IU is halted until the line fill is completed. If the IU executes a control transfer instruction (branch/CALL/JMPL/RETT/TRAP) during the line fill, the line fill will be terminated on the next fetch. If instruction burst fetch is enabled, instruction streaming is enabled even when the cache is disabled. In this case, the fetched instructions are only forwarded to the IU and the cache is not updated. During cache line refill, incremental burst are generated on the AHB bus.

If a memory access error occurs during a line fill with the IU halted, the corresponding valid bit in the cache tag will not be set. If the IU later fetches an instruction from the failed address, a cache miss will occur, triggering a new access to the failed address. If the error remains, an instruction access error trap (tt=0x1) will be generated.

55.3.2 Instruction cache tag

A instruction cache tag entry consists of several fields as shown in figure 177:

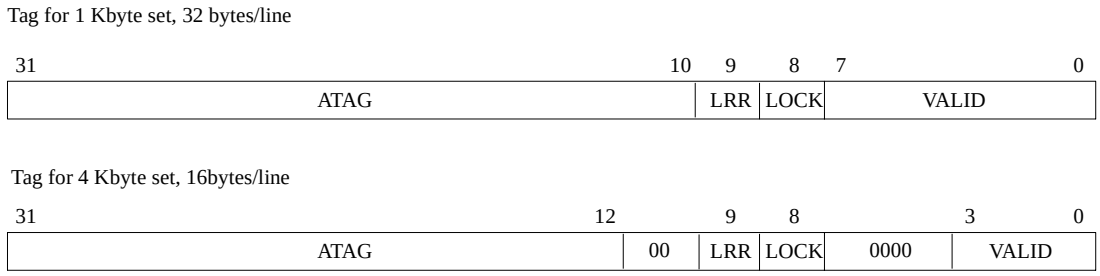


Figure 177. Instruction cache tag layout examples

Field Definitions:

[31:10]: Address Tag (ATAG) - Contains the tag address of the cache line.

[9]: LRR - Used by LRR algorithm to store replacement history, otherwise 0.

[8]: LOCK - Locks a cache line when set. 0 if cache locking not implemented.

[7:0]: Valid (V) - When set, the corresponding sub-block of the cache line contains valid data. These bits are set when a sub-block is filled due to a successful cache miss; a cache fill which results in a memory error will leave the valid bit unset. A FLUSH instruction will clear all valid bits. V[0] corresponds to address 0 in the cache line, V[1] to address 1, V[2] to address 2 and so on.

NOTE: only the necessary bits will be implemented in the cache tag, depending on the cache configuration. As an example, a 4 kbyte cache with 16 bytes per line would only have four valid bits and 20 tag bits. The cache rams are sized automatically by the ram generators in the model.

55.4 Data cache

55.4.1 Operation

The data cache can be configured as a direct-mapped cache or as a multi-set cache with associativity of 2 - 4 implementing either LRU or (pseudo-) random replacement policy or as 2-way associative cache implementing LRR algorithm. The set size is configurable to 1 - 64 kbyte and divided into cache lines of 16 - 32 bytes. Each line has a cache tag associated with it consisting of a tag field, valid field with one valid bit for each 4-byte sub-block and optional lock and LRR bits. On a data cache read-miss to a cachable location 4 bytes of data are loaded into the cache from main memory. The write policy for stores is write-through with no-allocate on write-miss. In a multi-set configuration a line to be replaced on read-miss is chosen according to the replacement policy. Locked AHB transfers are generated for LDST and SWAP instructions. If a memory access error occurs during a data load, the corresponding valid bit in the cache tag will not be set. and a data access error trap (tt=0x9) will be generated.

55.4.2 Write buffer

The write buffer (WRB) consists of three 32-bit registers used to temporarily hold store data until it is sent to the destination device. For half-word or byte stores, the stored data replicated into proper byte alignment for writing to a word-addressed device, before being loaded into one of the WRB registers. The WRB is emptied prior to a load-miss cache-fill sequence to avoid any stale data from being read in to the data cache.

Since the processor executes in parallel with the write buffer, a write error will not cause an exception to the store instruction. Depending on memory and cache activity, the write cycle may not occur until several clock cycles after the store instructions has completed. If a write error occurs, the currently executing instruction will take trap 0x2b.

Note: the 0x2b trap handler should flush the data cache, since a write hit would update the cache while the memory would keep the old value due the write error.

55.4.3 Data cache tag

A data cache tag entry consists of several fields as shown in figure 178:

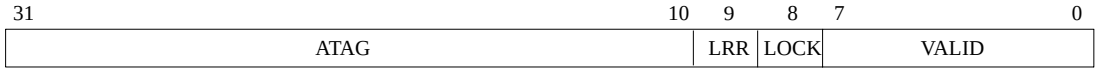


Figure 178. Data cache tag layout

Field Definitions:

[31:10]: Address Tag (ATAG) - Contains the address of the data held in the cache line.
[9]: LRR - Used by LRR algorithm to store replacement history. ‘0’ if LRR is not used.
[8]: LOCK - Locks a cache line when set. ‘0’ if instruction cache locking was not enabled in the configuration.
[3:0]: Valid (V) - When set, the corresponding sub-block of the cache line contains valid data. These bits are set when a sub-block is filled due to a successful cache miss; a cache fill which results in a memory error will leave the valid bit unset. V[0] corresponds to address 0 in the cache line, V[1] to address 1, V[2] to address 2 and V[3] to address 3.

NOTE: only the necessary bits will be implemented in the cache tag, depending on the cache configuration. As an example, a 2 kbyte cache with 32 bytes per line would only have eight valid bits and 21 tag bits. The cache rams are sized automatically by the ram generators in the model.

55.5 Additional cache functionality

55.5.1 Cache flushing

Both instruction and data cache are flushed by executing the FLUSH instruction. The instruction cache is also flushed by setting the FI bit in the cache control register, or by writing to any location with ASI=0x15. The data cache is also flushed by setting the FD bit in the cache control register, or by writing to any location with ASI=0x16. Cache flushing takes one cycle per cache line, during which the IU will not be halted, but during which the caches are disabled. When the flush operation is completed, the cache will resume the state (disabled, enabled or frozen) indicated in the cache control register. Diagnostic access to the cache is not possible during a FLUSH operation and will cause a data exception (trap=0x09) if attempted.

55.5.2 Diagnostic cache access

Tags and data in the instruction and data cache can be accessed through ASI address space 0xC, 0xD, 0xE and 0xF by executing LDA and STA instructions. Address bits making up the cache offset will be used to index the tag to be accessed while the least significant bits of the bits making up the address tag will be used to index the cache set.

Diagnostic read of tags is possible by executing an LDA instruction with ASI=0xC for instruction cache tags and ASI=0xE for data cache tags. A cache line and set are indexed by the address bits making up the cache offset and the least significant bits of the address bits making up the address tag. Similarly, the data sub-blocks may be read by executing an LDA instruction with ASI=0xD for instruction cache data and ASI=0xF for data cache data. The sub-block to be read in the indexed cache line and set is selected by A[4:2].

The tags can be directly written by executing a STA instruction with ASI=0xC for the instruction cache tags and ASI=0xE for the data cache tags. The cache line and set are indexed by the address bits making up the cache offset and the least significant bits of the address bits making up the address tag. D[31:10] is written into the ATAG field (see above) and the valid bits are written with the D[7:0] of

the write data. Bit D[9] is written into the LRR bit (if enabled) and D[8] is written into the lock bit (if enabled). The data sub-blocks can be directly written by executing a STA instruction with ASI=0xD for the instruction cache data and ASI=0xF for the data cache data. The sub-block to be read in the indexed cache line and set is selected by A[4:2].

In multi-way caches, the address of the tags and data of the ways are concatenated. The address of a tag or data is thus:

ADDRESS = WAY & LINE & DATA & “00”

Examples: the tag for line 2 in way 1 of a 2x4 Kbyte cache with 16 byte line would be:

A[13:12] = 1 (WAY)

A[11:5] = 2 (TAG)

=> TAG ADDRESS = 0x1040

The data of this line would be at addresses 0x1040 - 0x104C

55.5.3 Cache line locking

In a multi-set configuration the instruction and data cache controllers can be configured with optional lock bit in the cache tag. Setting the lock bit prevents the cache line to be replaced by the replacement algorithm. A cache line is locked by performing a diagnostic write to the instruction tag on the cache offset of the line to be locked setting the Address Tag field to the address tag of the line to be locked, setting the lock bit and clearing the valid bits. The locked cache line will be updated on a read-miss and will remain in the cache until the line is unlocked. The first cache line on certain cache offset is locked in the set 0. If several lines on the same cache offset are to be locked the locking is performed on the same cache offset and in sets in ascending order starting with set 0. The last set can not be locked and is always replaceable. Unlocking is performed in descending set order.

NOTE: Setting the lock bit in a cache tag and reading the same tag will show if the cache line locking was enabled during the LEON3 configuration: the lock bit will be set if the cache line locking was enabled otherwise it will be 0.

55.5.4 Local instruction ram

A local instruction ram can optionally be attached to the instruction cache controller. The size of the local instruction is configurable from 1-64 kB. The local instruction ram can be mapped to any 16 Mbyte block of the address space. When executing in the local instruction ram all instruction fetches are performed from the local instruction ram and will never cause IU pipeline stall or generate an instruction fetch on the AHB bus. Local instruction ram can be accessed through load/store integer word instructions (LD/ST). Only word accesses are allowed, byte, halfword or double word access to the local instruction ram will generate data exception.

55.5.5 Local scratch pad ram

Local scratch pad ram can optionally be attached to both instruction and data cache controllers. The scratch pad ram provides fast 0-waitstates ram memories for both instructions and data. The ram can be between 1 - 512 kbyte, and mapped on any 16 Mbyte block in the address space. Accessed performed to the scratch pad ram are not cached, and will not appear on the AHB bus. The scratch pads rams do not appear on the AHB bus, and can only be read or written by the processor. The instruction ram must be initialized by software (through store instructions) before it can be used. The default address for the instruction ram is 0x8e000000, and for the data ram 0x8f000000. See section 55.10 for additional configuration details. Note: local scratch pad ram can only be enabled when the MMU is disabled.

55.5.6 Data Cache snooping

To keep the data cache synchronized with external memory, cache snooping can be enabled through the *dsnoop* generic. When enabled, the data cache monitors write accesses on the AHB bus to cacheable locations. If an other AHB master writes to a cacheable location which is currently cached in the data cache, the corresponding cache line is marked as invalid.

55.5.7 Cache Control Register

The operation of the instruction and data caches is controlled through a common Cache Control Register (CCR) (figure 179). Each cache can be in one of three modes: disabled, enabled and frozen. If disabled, no cache operation is performed and load and store requests are passed directly to the memory controller. If enabled, the cache operates as described above. In the frozen state, the cache is accessed and kept in sync with the main memory as if it was enabled, but no new lines are allocated on read misses.

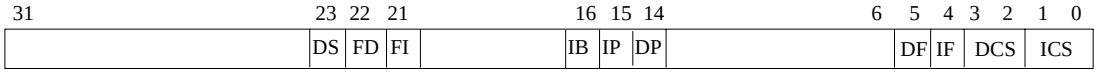


Figure 179. Cache control register

[23]: Data cache snoop enable [DS] - if set, will enable data cache snooping.

[22]: Flush data cache (FD). If set, will flush the instruction cache. Always reads as zero.

[21]: Flush Instruction cache (FI). If set, will flush the instruction cache. Always reads as zero.

[16]: Instruction burst fetch (IB). This bit enables burst fill during instruction fetch.

[15]: Instruction cache flush pending (IP). This bit is set when an instruction cache flush operation is in progress.

[14]: Data cache flush pending (DP). This bit is set when an data cache flush operation is in progress.

[5]: Data Cache Freeze on Interrupt (DF) - If set, the data cache will automatically be frozen when an asynchronous interrupt is taken.

[4]: Instruction Cache Freeze on Interrupt (IF) - If set, the instruction cache will automatically be frozen when an asynchronous interrupt is taken.

[3:2]: Data Cache state (DCS) - Indicates the current data cache state according to the following: X0= disabled, 01 = frozen, 11 = enabled.

[1:0]: Instruction Cache state (ICS) - Indicates the current data cache state according to the following: X0= disabled, 01 = frozen, 11 = enabled.

If the DF or IF bit is set, the corresponding cache will be frozen when an asynchronous interrupt is taken. This can be beneficial in real-time system to allow a more accurate calculation of worst-case execution time for a code segment. The execution of the interrupt handler will not evict any cache lines and when control is returned to the interrupted task, the cache state is identical to what it was before the interrupt. If a cache has been frozen by an interrupt, it can only be enabled again by enabling it in the CCR. This is typically done at the end of the interrupt handler before control is returned to the interrupted task.

55.5.8 Cache configuration registers

The configuration of the two caches if defined in two registers: the instruction and data configuration registers. These registers are read-only and indicate the size and configuration of the caches.

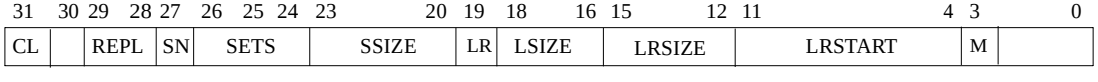


Figure 180. Cache configuration register

[31]: Cache locking (CL). Set if cache locking is implemented.

[29:28]: Cache replacement policy (REPL). 00 - no replacement policy (direct-mapped cache), 01 - least recently used (LRU), 10 - least recently replaced (LRR), 11 - random

[27]: Cache snooping (SN). Set if snooping is implemented.

[26:24]: Cache associativity (SETS). Number of sets in the cache: 000 - direct mapped, 001 - 2-way associative, 010 - 3-way associative, 011 - 4-way associative

[23:20]: Set size (SSIZE). Indicates the size (Kbytes) of each cache set. Size = 2^{SIZE}

[19]: Local ram (LR). Set if local scratch pad ram is implemented.

[18:16]: Line size (LSIZE). Indicated the size (words) of each cache line. Line size = 2^{LSZ}

[15:12]: Local ram size (LRSZ). Indicates the size (Kbytes) of the implemented local scratch pad ram. Local ram size = 2^{LRSZ}

[11:4]: Local ram start address. Indicates the 8 most significant bits of the local ram start address.

[3]: MMU present. This bit is set to '1' if an MMU is present.

All cache registers are accessed through load/store operations to the alternate address space (LDA/STA), using ASI = 2. The table below shows the register addresses:

*Table 627.*ASI 2 (system registers) address map

Address	Register
0x00	Cache control register
0x04	Reserved
0x08	Instruction cache configuration register
0x0C	Data cache configuration register

55.5.9 Software consideration

After reset, the caches are disabled and the cache control register (CCR) is 0. Before the caches may be enabled, a flush operation must be performed to initialized (clear) the tags and valid bits. A suitable assembly sequence could be:

```
flush
set 0x81000f, %g1
sta%g1, [%g0] 2
```

55.6 Memory management unit

A memory management unit (MMU) compatible with the SPARC V8 reference MMU can optionally be configured. For details on operation, see the SPARC V8 manual.

55.6.1 ASI mappings

When the MMU is used, the following ASI mappings are added:

*Table 628.*MMU ASI usage

ASI	Usage
0x10	Flush page
0x10	MMU flush page
0x13	MMU flush context
0x14	MMU diagnostic dcache context access
0x15	MMU diagnostic icache context access
0x19	MMU registers
0x1C	MMU bypass
0x1D	MMU diagnostic access
0x1E	MMU snoop tags diagnostic access

55.6.2 MMU/Cache operation

When the MMU is disabled, the caches operate as normal with physical address mapping. When the MMU is enabled, the caches tags store the virtual address and also include an 8-bit context field. If cache snooping is desired, bit 2 of the *dsnoop* generic has to be set. This will store the physical tag in a separate RAM block, which then is used for snooping. In addition, the size of each data cache way has to be smaller or equal to 4 kbyte (MMU page size). This is necessary to avoid aliasing in the cache since the virtual tags are indexed with a virtual offset while the physical tags are indexed with a physical offset. Physical tags and snoop support is needed for SMP systems using the MMU (linux-2.6).

Because the cache is virtually tagged, no extra clock cycles are needed in case of a cache load hit. In case of a cache miss or store hit (write-through cache) at least 2 extra clock cycles are used if there is a TLB hit. If there is a TLB miss the page table must be traversed, resulting in up to 4 AMBA read accesses and one possible writeback operation. If a combined TLB is used by the instruction cache, the translation is stalled until the TLB is free. If fast TLB operation is selected (tlb_type = 2), the TLB will be accessed simultaneously with tag access, saving 2 clocks on cache miss. This will increase the area somewhat, and may reduce the timing, but usually results in better overall throughput.

55.6.3 MMU registers

The following MMU registers are implemented:

*Table 629.*MMU registers (ASI = 0x19)

Address	**Register**
0x000	MMU control register
0x100	Context pointer register
0x200	Context register
0x300	Fault status register
0x400	Fault address register

The definition of the registers can be found in the SPARC V8 manual.

55.6.4 Translation look-aside buffer (TLB)

The MMU can be configured to use a shared TLB, or separate TLB for instructions and data. The number of TLB entries can be set to 2 - 32 in the configuration record. The organisation of the TLB and number of entries is not visible to the software and does thus not require any modification to the operating system.

55.6.5 Snoop tag diagnostic access

If the MMU has been configured to use separate snoop tags, they can be accessed via ASI 0x1E. This is primarily useful for RAM testing, and should not be performed during normal operation. The figure below shows the layout of the snoop tag for a 1 Kbyte data cache:

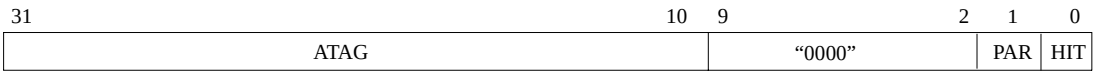


Figure 181. Snoop cache tag layout

[31:10] Address tag. The physical address tag of the cache line.
[1]: Parity. The odd parity over the data tag. LEON3FT only.
[0]: Snoop hit. When set, the cache line is not valid and will cause a cache miss if accessed by the processor.

55.6.6 Variable page sizes

The standard page size for the SRMMU is 4k. The page size can be changed to 8k, 16k or 32k. The indexing for 4k page size is [8,6,6] bits for each level. For 8k pages it is [7,6,6], for 16k pages it is [6,6,6] and finally for 32k pages it is [4,7,6]. The layouts of the indexes are choosen so that PTE pagetables can be joined together inside one page without leaving holes.This is needed for Linux to function. The page size can also be choosen by the program at runtime if selecting “programmable” in the GUI configurator (setting generic *mmupgsz* to 1). In this case the page size is choosen by bit [17-16] in the MMU ctrl register: 0 for 4k, 1 for 8k, 2 for 16k and 3 for 32k. In Linux the page size have to be choosen with the option *CONFIG_PAGE_SIZE_LEON* in the GUI configurator.

55.7 Floating-point unit and custom co-processor interface

The SPARC V8 architecture defines two (optional) co-processors: one floating-point unit (FPU) and one user-defined co-processor. The LEON3 pipeline provides an interface port for both of these units. Two different FPU’s can be interfaced: Gaisler Research’s GRFPU, and the Meiko FPU from Sun. Selection of which FPU to use is done through the VHDL model’s generic map. The characteristics of the FPU’s are described in the next sections.

55.7.1 Gaisler Research’s floating-point unit (GRFPU)

The high-performance GRFPU operates on single- and double-precision operands, and implements all SPARC V8 FPU instructions. The FPU is interfaced to the LEON3 pipeline using a LEON3-specific FPU controller (GRFPC) that allows FPU instructions to be executed simultaneously with integer instructions. Only in case of a data or resource dependency is the integer pipeline held. The GRFPU is fully pipelined and allows the start of one instruction each clock cycle, with the exception is FDIV and FSQRT which can only be executed one at a time. The FDIV and FSQRT are however executed in a separate divide unit and do not block the FPU from performing all other operations in parallel.

All instructions except FDIV and FSQRT has a latency of three cycles, but to improve timing, the LEON3 FPU controller inserts an extra pipeline stage in the result forwarding path. This results in a latency of four clock cycles at instruction level. The table below shows the GRFPU instruction timing when used together with GRFPC:

*Table 630.*GRFPU instruction timing with GRFPC

Instruction	Throughput	Latency
FADDS, FADDD, FSUBS, FSUBD,FMULS, FMULD, FSMULD, FITOS, FITOD, FSTOI, FDTOI, FSTOD, FDTOS, FCMPS, FCMPD, FCMPES. FCMPED	1	4
FDIVS	14	16
FDIVD	15	17
FSQRTS	22	24
FSQRTD	23	25

The GRFPC controller implements the SPARC deferred trap model, and the FPU trap queue (FQ) can contain up to 7 queued instructions when an FPU exception is taken. When the GRFPU is enabled in the model, the version field in %fsr has the value of 2.

55.7.2 GRFPU-Lite

GRFPU-Lite is a smaller version of GRFPU, suitable for FPGA implementations with limited logic resources. The GRFPU-Lite is not pipelined and executes thus only one instruction at a time. To improve performance, the FPU controller (GRLFPC) allows GRFPU-Lite to execute in parallel with

the processor pipeline as long as no new FPU instructions are pending. Below is a table of worst-case throughput of the GRFPU-Lite:

*Table 631.*GRFPU-Lite worst-case instruction timing with GRLFPC

Instruction	Throughput	Latency
FADDS, FADDD, FSUBS, FSUBD,FMULS, FMULD, FSMULD, FITOS, FITOD, FSTOI, FDTOI, FSTOD, FDTOS, FCMPS, FCMPD, FCMPES. FCMPED	8	8
FDIVS	31	31
FDIVD	57	57
FSQRTS	46	46
FSQRTD	65	65

When the GRFPU-Lite is enabled in the model, the version field in %fsr has the value of 3.

55.7.3 The Meiko FPU

The Meiko floating-point core operates on both single- and double-precision operands, and implements all SPARC V8 FPU instructions. The Meiko FPU is interfaced through the Meiko FPU controller (MFC), which allows one FPU instruction to execute in parallel with IU operation. The MFC implements the SPARC deferred trap model, and the FPU trap queue (FQ) can contain up to one queued instruction when an FPU exception is taken.

When the Meiko FPU is enabled in the model, the version field in %fsr has the value of 1.

The Meiko FPU is not distributed by Gaisler Research, and must be obtained separately from Sun.

55.7.4 Generic co-processor

LEON can be configured to provide a generic interface to a user-defined co-processor. The interface allows an execution unit to operate in parallel to increase performance. One co-processor instruction can be started each cycle as long as there are no data dependencies. When finished, the result is written back to the co-processor register file.

55.8 Vendor and device identifiers

The core has vendor identifiers 0x01 (Gaisler Research) and device identifiers 0x003. For description of vendor and device identifiers see GRLIB IP Library User's Manual.

55.9 Implementation

55.9.1 Area and timing

Both area and timing of the LEON3 core depends strongly on the selected configuration, target technology and the used synthesis tool. The table below indicates the typical figures for two baseline configurations.

*Table 632.*Area and timing

	Actel AX2000			ASIC (0.13 um)	
Configuration	Cells	RAM64	MHz	Gates	MHz
LEON3, 8 + 8 Kbyte cache	6,500	40	30	25,000	400
LEON3, 8 + 8 Kbyte cache + DSU3	7,500	40	25	30,000	400

55.9.2 Technology mapping

LEON3 has two technology mapping generics, *fabtech* and *memtech*. The *fabtech* generic controls the implementation of some pipeline features, while *memtech* selects which memory blocks will be used to implement cache memories and the IU/FPU register file. *Fabtech* can be set to any of the provided technologies (0 - NTECH) as defined in the GRPIB.TECH package. See the GRLIB Users's Manual for available settings for *memtech*.

55.9.3 RAM usage

The LEON3 core maps all usage of RAM memory on the *syncram*, *syncram_2p* and *syncram_dp* components from the technology mapping library (TECHMAP). The type, configuration and number of RAM blocks is described below.

Register file

The register file is implemented with two *synram_2p* blocks for all technologies where the *regfile_3p_infer* constant in TECHMAP.GENCOMP is set to 0. The organization of the syncram_2p is shown in the following table:

*Table 633.*syncram_2p sizes for LEON3 register file

Register windows	Syncram_2p organization
2 - 3	64x32
4 - 7	128x32
8 - 15	256x32
16-31	512x31
32	1024x32

If *regfile_3p_infer* is set to 1, the synthesis tool will automatically infer the register. On FPGA technologies, it can be in either flip-flops or RAM cells, depending on the tool and technology. On ASIC technologies, it will be flip-flops. The amount of flip-flops inferred is equal to the number of registers:

Number of flip-flops = ((NWINDOWS *16) + 8) * 32

FP register file

If FPU support is enabled, the FP register file is implemented with four *synram_2p* blocks when the *regfile_3p_infer* constant in TECHMAP.GENCOMP is set to 0. The organization of the syncram_2p blocks is 16x32.

If *regfile_3p_infer* is set to 1, the synthesis tool will automatically infer the FP register file. For ASIC technologies the number of inferred flip-flops is equal to number of bits in the FP register file which is 32 * 32 = 1024.

Cache memories

RAM blocks are used to implement the cache tags and data memories. Depending on cache configuration, different types and sizes of RAM blocks are used.

The tag memory is implemented with one *syncram* per cache way when no snooping is enabled. The tag memory depth and width is calculated as follows:

Depth = (cache way size in bytes) / (cache line size in bytes)

Width = 32 - log2(cache way size in bytes) + (cache line size in bytes)/4 + lrr + lock

For a 2 Kbyte cache way with lrr replacement and 32 bytes/line, the tag RAM depth will be (2048/32) = 64. The width will be: 32 - log2(2048) + 32/4 + 1 = 32 - 11 + 8 + 1 = 28. The tag RAM organization

will thus be 64x28 for the configuration. If the MMU is enabled, the tag memory width will increase with 8 to store the process context ID, and the above configuration will us a 64x36 RAM.

If snooping is enabled, the tag memories will be implemented using the *syncram_dp* component (dual-port RAM). One port will be used by the processor for cache access/refill, while the other port will be used by the snooping and invalidation logic. The size of the *syncram_dp* block will be the same as when snooping is disabled. If physical snooping is enabled (separate snoop tags), one extra RAM block per data way will be instatiated to hold the physical tags. The width of the RAM block will be the same as the tag address: 32 - log2(way size). A 4 Kbyte data cache way will thus require a 32 - 12 = 20 bit wide RAM block for the physical tags. If fast snooping is enabled, the tag RAM (virtual and physical) will be implemented using *syncram_2p* instead of *syncram_dp*. This can be used to implement snooping on technologies which lack dual-port RAM but have 2-port RAM.

The data part of the caches (storing instructions or data) is always 32 bit wide. The depth is equal to the way size in bytes, divided by 4. A cache way of 2 Kbyte will thus use *syncram* component with and organization of 512x32.

Instruction Trace buffer

The instruction trace buffer will use four identical RAM blocks (*syncram*) to implement the buffer memory. The syncrams will always be 32-bit wide. The depth will depend on the TBUF generic, which indicates the total size of trace buffer in Kbytes. If TBUF = 1 (1 Kbyte), then four RAM blocks of 64x32 will be used. If TBUF = 2, then the RAM blocks will be 128x32 and so on.

Scratch pad RAM

If the instruction scratch pad RAM is enabled, a *syncram* block will be instantiated with a 32-bit data width. The depth of the RAM will correspond to the configured scratch pad size. An 8 Kbyte scratch pad will use a *syncram* with 2048x32 organization. The RAM block for the data scratch pad will be configured in the same way as the instruction scratch pad.

55.9.4 Double clocking

The LEON3 CPU core be clocked at twice the clock speed of the AMBA AHB bus. When clocked at double AHB clock frequency, all CPU core parts including integer unit and caches will operate at double AHB clock frequency while the AHB bus access is performed at the slower AHB clock frequency. The two clocks have to be synchronous and a multicycle paths between the two clock domains have to be defined at synthesis tool level. A separate component (leon3s2x) is provided for the double clocked core. Double clocked versions of DSU (dsu3_2x) and MP interrupt controller (irqmp2x) are used in a double clocked LEON3 system. An AHB clock qualifier signal (*clken* input) is used to identify end of AHB cycle. The AHB qualifier signal is generated in CPU clock domain and is high during the last CPU clock cycle under AHB clock low-phase. Sample *leon3-clk2x* design provides a module that generates an AHB clock qualifier signal.

Double-clocked design has two clock domains: AMBA clock domains (HCLK) and CPU clock domain (CPUCLK). LEON3 (leon3s2x component) and DSU3 (dsu3_2x) belong to CPU clock domain (clocked by CPUCLK), while the rest of the system is in AMBA clock domain (clocked by HCLK). Paths between the two clock domains (paths starting in CPUCLK domain and ending in

HCLK and paths starting in HCLK domain and ending in CPUCLK domain) are multicycle paths with propagation time of two CPUCLK periods (or one HCLK period) with following exceptions:

Start point	Through	End point	Propagation time
leon3s2x core			
CPUCLK	ahbi	CPUCLK	2 CPUCLK
CPUCLK	ahbsi	CPUCLK	2 CPUCLK
CPUCLK	ahbso	CPUCLK	2 CPUCLK
HCLK	irqi	CPUCLK	1 CPUCLK
CPUCLK	irqo	HCLK	1 CPUCLK
CPUCLK		u0_0/p0/c0/sync0/r[*] (register)	1 CPUCLK

Start point	Through	End point	Propagation time
dsu3_2x core			
CPUCLK	ahbmi	CPUCLK	2 CPUCLK
CPUCLK	ahbsi	CPUCLK	2 CPUCLK
	dsui	CPUCLK	1 CPUCLK
r[*] (register)		rh[*] (register)	1 CPUCLK
irqmp2x core			
r2[*] (register)		r[*] (register)	1 CPUCLK

55.9.5 Clock gating

To further reduce the power consumption of the processor, the clock can be gated-off when the processor has entered power-down state. Since the cache controllers and MMU operate in parallel with the processor, the clock cannot be gated immediately when the processor has entered the power-down state. Instead, a power-down signal (DBGO.idle) is generated when all outstanding AHB accesses have been completed and it is safe to gate the clock. This signal should be clocked though a positive-edge flip-flop followed by a negative-edge flip-flop to guarantee that the clock is gated off during the clock-low phase. To insure proper start-up state, the clock should not be gated during reset.

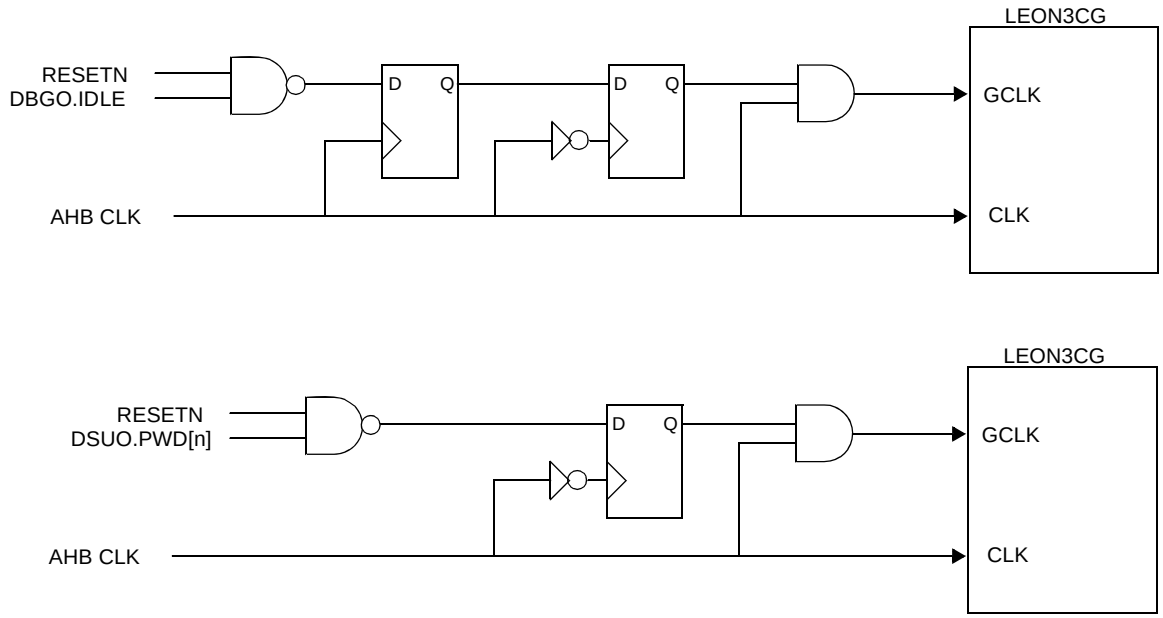


Figure 182. Examples of LEON3 clock gating

The processor should exit the power-down state when an interrupt become pending. The signal DBGO.ipend will then go high when this happen, and should be used to re-enable the clock.

When the debug support unit (DSU3) is used, the DSUO.pwd signal should be used instead of DBGO.idle. This will insure that the clock also is re-enabled when the processor is switched from power-down to debug state by the DSU. The DSUO.pwd is a vector with one power-down signal per CPU (for SMP systems). DSUO.pwd takes DBGO.ipend into account, and no further gating or latching needs to be done of this signal. If cache snooping has been enabled, the continuous clock will insure that the snooping logic is activated when necessary and will keep the data cache synchronized even when the processor clock is gated-off. In a multi-processor system, all processor except node 0 will enter power-down after reset and will allow immediate clock-gating without additional software support.

Clock-tree routing must insure that the continuous clock (CLK) and the gated clock (GCLK) are phase-aligned. The template design *leon3-clock-gating* shows an example of a clock-gated system. The *leon3cg* entity should be used when clock gating is implemented. This entity has one input more (GCLK) which should be driven by the gated clock. Using the double-clocked version of leon3 (leon3s2x), the GCLK2 is the gated double-clock while CLK and CLK2 should be continuous.

55.9.6 Scan support

If the SCANTEST generic is set to 1, support for scan testing is enabled. This will make use of the AHB scan support signals in the following manner: when AHBI.testen and AHBI.scanen are both ‘1’, the select signals to all RAM blocks (cache RAM, register file and DSU trace buffers) are disabled. This means that when the scan chain is shifted, no accidental write or read can occur in the RAM blocks. The scan signal AHBI.testrst is not used as there are no asynchronous resets in the LEON3 core.

55.10 Configuration options

Table 634 shows the configuration options of the core (VHDL generics).

*Table 634.*Configuration options

Generic	Function	Allowed range	Default
hindex	AHB master index	0 - NAHBMST-1	0
fabtech	Target technology	0 - NTECH	0 (inferred)
memtech	Vendor library for regfile and cache RAMs	0 - NTECH	0 (inferred)
nwindows	Number of SPARC register windows. Choose 8 windows to be compatible with Bare-C and RTEMS cross-compilers.	2 - 32	8
dsu	Enable Debug Support Unit interface	0 - 1	0
fpu	Floating-point Unit 0 : no FPU 1 - 7: GRFPU 1 - inferred multiplier, 2 - DW multiplier, 3 - Module Generator multiplier 8 - 14: GRFPU-Lite 8 - simple FPC, 9 - data forwarding FPC, 10 - non-blocking FPC 15: Meiko 16 - 31: as above (modulo 16) but use netlist	0 - 31	0
v8	Generate SPARC V8 MUL and DIV instructions 0 : No multiplier or divider 1 : 16x16 multiplier 2: 16x16 pipelined multiplier 16#32# : 32x32 pipelined multiplier	0 - 16#3F#	0
cp	Generate co-processor interface	0 -1	0
mac	Generate SPARC V8e SMAC/UMAC instruction	0 - 1	0
pclow	Least significant bit of PC (Program Counter) that is actually generated. PC[1:0] are always zero and are normally not generated. Generating PC[1:0] makes VHDL-debugging easier.	0, 2	2
notag	Currently not used	-	-
nwp	Number of watchpoints	0 - 4	0
icen	Enable instruction cache	0 - 1	1
irepl	Instruction cache replacement policy. 0 - least recently used (LRU), 1 - least recently replaced (LRR), 2 - random	0 - 1	0
isets	Number of instruction cache sets	1 - 4	1
ilinesize	Instruction cache line size in number of words	4, 8	4
isetsize	Size of each instruction cache set in kByte	1 - 256	1
isetlock	Enable instruction cache line locking	0 - 1	0
dcen	Data cache enable	0 - 1	1
drepl	Data cache replacement policy. 0 - least recently used (LRU), 1 - least recently replaced (LRR), 2 - random	0 - 1	0
dsets	Number of data cache sets	1 - 4	1
dlinesize	Data cache line size in number of words	4, 8	4
dsetsize	Size of each data cache set in kByte	1 - 256	1
dsetlock	Enable data cache line locking	0 - 1	0

*Table 634.*Configuration options

Generic	Function	Allowed range	Default
dsnoop	Enable data cache snooping Bit 0-1: 0: disable, 1: slow, 2: fast (see text) Bit 2: 0: simple snooping, 1: save extra physical tags (MMU snooping)	0 - 6	0
ilram	Enable local instruction RAM	0 - 1	0
ilramsize	Local instruction RAM size in kB	1 - 512	1
ilramstart	8 MSB bits used to decode local instruction RAM area	0 - 255	16#8E#
dlram	Enable local data RAM (scratch-pad RAM)	0 - 1	0
dlramsize	Local data RAM size in kB	1 - 512	1
dlramstart	8 MSB bits used to decode local data RAM area	0 - 255	16#8F#
mmuen	Enable memory management unit (MMU)	0 - 1	0
itlbnum	Number of instruction TLB entries	2 - 64	8
dtlbnum	Number of data TLB entries	2 - 64	8
tlb_type	0 : separate TLB with slow write 1: shared TLB with slow write 2: separate TLB with fast write	0 - 2	1
tlb_rep	LRU (0) or Random (1) TLB replacement	0 - 1	0
lddel	Load delay. One cycle gives best performance, but might create a critical path on targets with slow (data) cache memories. A 2-cycle delay can improve timing but will reduce performance with about 5%.	1 - 2	2
disas	Print instruction disassembly in VHDL simulator console.	0 - 1	0
tbuf	Size of instruction trace buffer in kB (0 - instruction trace disabled)	0 - 64	0
pwd	Power-down. 0 - disabled, 1 - area efficient, 2 - timing efficient.	0 - 2	1
svt	Enable single-vector trapping	0 - 1	0
rstaddr	Default reset start address	0 - (2**^{20}-1)	0
smp	Enable multi-processor support	0 - 15	0
cached	Fixed cacheability mask	0 - 16#FFFF#	0
scantest	Enable scan test support	0 - 1	0
mmupgsz	Specify page size. 0: 4k 1: programmable through bit [17-16] of the MMU ctrl register 2: 4k 3: 8k 4:16k 5:32k for programmable page size the following mapping for bit [17-16] of the MMU ctrl register apply: 0: 4k 1: 8k 2:16k 3:32k	0-5	0

55.11 Signal descriptions

Table 635 shows the interface signals of the core (VHDL ports).

*Table 635.*Signal descriptions

Signal name	Field	Type	Function	Active
CLK	N/A	Input	AMBA and processor clock (leon3s, leon3cg)	-
CLK2		Input	Processor clock in 2x mode (leon3sx2)	
GCLK2		Input	Gated processor clock in 2x mode (leon3sx2)	
RSTN	N/A	Input	Reset	Low
AHBI	*	Input	AHB master input signals	-
AHBO	*	Output	AHB master output signals	-
AHBSI	*	Input	AHB slave input signals	-
IRQI	IRL[3:0]	Input	Interrupt level	High
	RST	Input	Reset power-down and error mode	High
	RUN	Input	Start after reset (SMP system only)	
IRQO	INTACK	Output	Interrupt acknowledge	High
	IRL[3:0]	Output	Processor interrupt level	High
DBGI	-	Input	Debug inputs from DSU	-
DBGO	-	Output	Debug outputs to DSU	-
	ERROR		Processor in error mode, execution halted	Low
GCLK		Input	Gated processor clock for leon3cg	

* see GRLIB IP Library User's Manual

55.12 Library dependencies

Table 636 shows the libraries used when instantiating the core (VHDL libraries).

*Table 636.*Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AHB signal definitions
GAISLER	LEON3	Component, signals	LEON3 component declaration, interrupt and debug signals declaration

55.13 Component declaration

The core has the following component declaration.

```
entity leon3s is
  generic (
    hindex    : integer                := 0;
    fabtech   : integer range 0 to NTECH  := 0;
    memtech   : integer range 0 to NTECH  := 0;
    nwindows  : integer range 2 to 32 := 8;
    dsu       : integer range 0 to 1  := 0;
    fpu       : integer range 0 to 3  := 0;
    v8        : integer range 0 to 2  := 0;
    cp        : integer range 0 to 1  := 0;
    mac       : integer range 0 to 1  := 0;
    pclow     : integer range 0 to 2  := 2;
    notag     : integer range 0 to 1  := 0;
    nwp       : integer range 0 to 4  := 0;
    icen      : integer range 0 to 1  := 0;
```

```
    irepl     : integer range 0 to 2  := 2;
    isets     : integer range 1 to 4  := 1;
    ilinesize : integer range 4 to 8  := 4;
    isetsize  : integer range 1 to 256 := 1;
    isetlock  : integer range 0 to 1  := 0;
    dcen      : integer range 0 to 1  := 0;
    drepl     : integer range 0 to 2  := 2;
    dsets     : integer range 1 to 4  := 1;
    dlinesize : integer range 4 to 8  := 4;
    dsetsize  : integer range 1 to 256 := 1;
    dsetlock  : integer range 0 to 1  := 0;
    dsnoop    : integer range 0 to 6:= 0;
    ilram      : integer range 0 to 1 := 0;
    ilramsize  : integer range 1 to 512 := 1;
    ilramstart : integer range 0 to 255 := 16#8e#;
    dlram      : integer range 0 to 1 := 0;
    dlramsize  : integer range 1 to 512 := 1;
    dlramstart : integer range 0 to 255 := 16#8f#;
    mmuen     : integer range 0 to 1  := 0;
    itlbnum   : integer range 2 to 64 := 8;
    dtlbnum   : integer range 2 to 64 := 8;
    tlb_type  : integer range 0 to 1 := 1;
    tlb_rep   : integer range 0 to 1 := 0;
    lddel     : integer range 1 to 2  := 2;
    disas     : integer range 0 to 1  := 0;
    tbuf      : integer range 0 to 64 := 0;
    pwd       : integer range 0 to 2  := 2;     -- power-down
    svt       : integer range 0 to 1  := 1;     -- single vector trapping
    rstaddr   : integer               := 0;
    smp       : integer range 0 to 15 := 0;    -- support SMP systems
    cached    : integer               := 0;     -- cacheability table
    scantest  : integer               := 0
  );

port (
    clk    : in  std_ulogic;
    rstn   : in  std_ulogic;
    ahbi   : in  ahb_mst_in_type;
    ahbo   : out ahb_mst_out_type;
    ahbsi  : in  ahb_slv_in_type;
    ahbso  : in  ahb_slv_out_vector;
    irqi   : in  l3_irq_in_type;
    irqo   : out l3_irq_out_type;
    dbgi   : in  l3_debug_in_type;
    dbgo   : out l3_debug_out_type
  );
end;
```

Beilage 4

AHBCTRL GRLIB Auszug

4 AHBCTRL - AMBA AHB controller with plug&play support

4.1 Overview

The AMBA AHB controller is a combined AHB arbiter, bus multiplexer and slave decoder according to the AMBA 2.0 standard.

The controller supports up to 16 AHB masters, and 16 AHB slaves. The maximum number of masters and slaves are defined in the GRLIB.AMBA package, in the VHDL constants NAHBSLV and NAHBMST. It can also be set with the *nahbm* and *nahbs* VHDL generics.

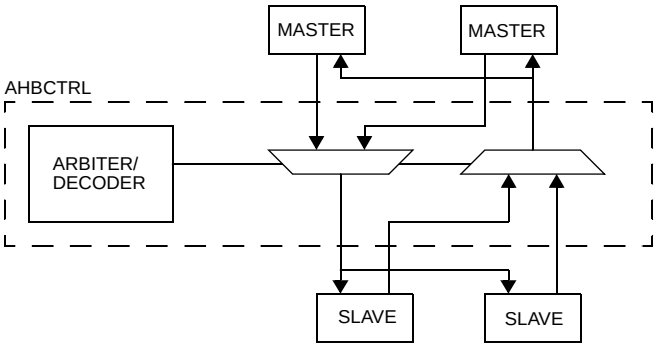


Figure 3. AHB controller block diagram

4.2 Operation

4.2.1 Arbitration

The AHB controller supports two arbitration algorithms: fixed-priority and round-robin. The selection is done by the VHDL generic *rrobin*. In fixed-priority mode (*rrobin* = 0), the bus request priority is equal to the master's bus index, with index 0 being the lowest priority. If no master requests the bus, the master with bus index 0 (set by the VHDL generic *defmast)* will be granted.

In round-robin mode, priority is rotated one step after each AHB transfer. If no master requests the bus, the last owner will be granted (bus parking). The VHDL generic *mprio* can be used to specify one or more masters that should be prioritized when the core is configured for round-robin mode.

During incremental bursts, the AHB master should keep the bus request asserted until the last access as recommended in the AMBA 2.0 specification, or it might loose bus ownership. For fixed-length burst, the AHB master will be granted the bus during the full burst, and can release the bus request immediately after the first access has started. For this to work however, the VHDL generic *fixbrst* should be set to 1.

4.2.2 Decoding

Decoding (generation of HSEL) of AHB slaves is done using the plug&play method explained in the GRLIB User's Manual. A slave can occupy any binary aligned address space with a size of 1 - 4096 Mbyte. A specific I/O area is also decoded, where slaves can occupy 256 byte - 1 Mbyte. The default address of the I/O area is 0xFFF00000, but can be changed with the *ioaddr* and *iomask* VHDL generics. Access to unused addresses will cause an AHB error response.

4.2.3 Plug&play information

GRLIB devices contain a number of plug&play information words which are included in the AHB records they drive on the bus (see the GRLIB user's manual for more information). These records are combined into an array which is connected to the AHB controller unit.

The plug&play information is mapped on a read-only address area, defined by the *cfgaddr* and *cfgmask* VHDL generics, in combination with the *ioaddr* and *iomask* VHDL generics. By default, the area is mapped on address 0xFFFFF000 - 0xFFFFFFFF. The master information is placed on the first 2 kbyte of the block (0xFFFFF000 - 0xFFFFF800), while the slave information is placed on the second 2 kbyte block. Each unit occupies 32 bytes, which means that the area has place for 64 masters and 64 slaves. The address of the plug&play information for a certain unit is defined by its bus index. The address for masters is thus 0xFFFFF000 + n*32, and 0xFFFFF800 + n*32 for slaves.

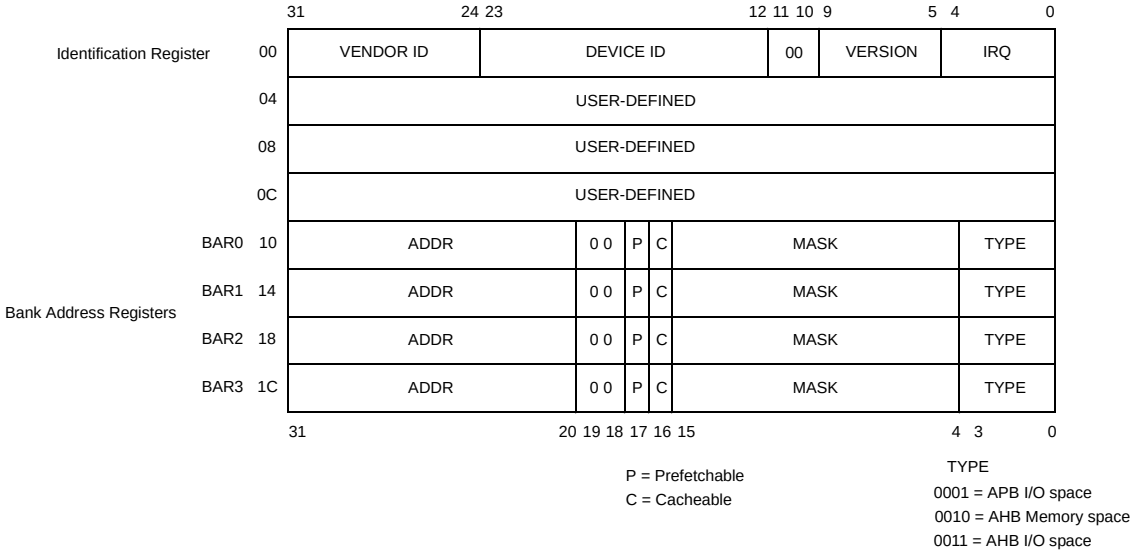


Figure 4. AHB plug&play information record

4.3 AHB split support

AHB SPLIT functionality is supported if the *split* VHDL generic is set to 1. In this case, all slaves must drive the AHB SPLIT signal.

It is important to implement the split functionality in slaves carefully since locked splits can otherwise easily lead to deadlocks. A locked access to a slave which is currently processing (it has returned a split response but not yet split complete) an access which it returned split for to another master must be handled first. This means that the slave must either be able to return an OKAY response to the locked access immediately or it has to split it but return split complete to the master performing the locked transfer before it has finished the first access which received split.

4.4 AHB bus monitor

An AHB bus monitor is integrated into the core. It is enabled with the *enbusmon* generic. It has the same functionality as the AHB and arbiter parts in the AMBA monitor core (AMBAMON). For more information on which rules are checked se the AMBAMON documentation.

4.5 AHB early burst termination

The core will perform early burst termination if the VHDL generic *enebterm* is set to 1. The core will agressively re-arbitrate all transfers which are not locked. This functionality is only intended to be used, and can only be enabled, in simulation.

4.6 Registers

The core does not implement any registers.

4.7 Configuration options

Table 24 shows the configuration options of the core (VHDL generics).

Table 24. Configuration options

Generic	Function	Allowed range	Default
ioaddr	The MSB address of the I/O area. Sets the 12 most significant bits in the 32-bit AHB address (i.e. 31 downto 20)	0 - 16#FFF#	16#FFF#
iomask	The I/O area address mask. Sets the size of the I/O area and the start address together with ioaddr.	0 - 16#FFF#	16#FFF#
cfgaddr	The MSB address of the configuration area. Sets 12 bits in the 32-bit AHB address (i.e. 19 downto 8).	0 - 16#FFF#	16#FF0#
cfgmask	The address mask of the configuration area. Sets the size of the configuration area and the start address together with cfgaddr. If set to 0, the configuration will be disabled.	0 - 16#FFF#	16#FF0#
rrobin	Selects between round-robin (1) or fixed-priority (0) bus arbitration algorithm.	0 - 1	0
split	Enable support for AHB SPLIT response	0 - 1	0
defmast	Default AHB master	0 - NAHBMST-1	0
ioen	AHB I/O area enable. Set to0 to disable the I/O area	0 - 1	1
nahbm	Number of AHB masters	1 - NAHBMST	NAHBMST
nahbs	Number of AHB slaves	1 - NAHBSLV	NAHBSLV
timeout	Perform bus timeout checks (NOT IMPLEMENTED).	0 - 1	0
fixbrst	Enable support for fixed-length bursts	0 - 1	0
debug	Print configuration (0=none, 1=short, 2=all cores)	0 - 2	2
fpnpen	Enables full decoding of the PnP configuration records. When disabled the user-defined registers in the PnP configuration records are not mapped in the configuration area.	0 - 1	0
icheck	Check bus index	0 - 1	1
devid	Assign unique device identifier readable from plug and play area.	N/A	0
enbusmon	Enable AHB bus monitor	0 - 1	0
assertwarn	Enable assertions for AMBA recommendations. Violations are asserted with severity warning.	0 - 1	0
asserterr	Enable assertions for AMBA requirements. Violations are asserted with severity error.	0 - 1	0
hmstdisable	Disable AHB master rule check. To disable a master rule check a value is assigned so that the binary representation contains a one at the position corresponding to the rule number, e.g 0x80 disables rule 7.	N/A	0
hslvdisable	Disable AHB slave tests. Values are assigned as for hmstdisable.	N/A	0
arbdisable	Disable Arbiter tests. Values are assigned as for hmstdisable.	N/A	0
mprio	Master(s) with highest priority. This value is converted to a vector where each position corresponds to a master. To prioritize masters x and y set this generic to $2^x + 2^y$.	N/A	0
enebterm	Enable early burst termination, only in simulation.	0 - 1	0

4.8 Signal descriptions

Table 25 shows the interface signals of the core (VHDL ports).

Table 25. Signal descriptions

Signal name	Field	Type	Function	Active
RST	N/A	Input	AHB reset	Low
CLK	N/A	Input	AHB clock	-
MSTI	*	Output	AMBA AHB master interface record array	-
MSTO	*	Input	AMBA AHB master interface record array	-
SLVI	*	Output	AMBA AHB slave interface record array	-
SLVO	*	Input	AMBA AHB slave interface record array	-

* see GRLIB IP Library User's Manual

4.9 Library dependencies

Table 26 shows libraries used when instantiating the core (VHDL libraries).

Table 26. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Types	AMBA signal type definitions

4.10 Component declaration

```
library grlib;
use grlib.amba.all;

component ahbctrl
  generic (
    defmast : integer := 0;-- default master
    split   : integer := 0;-- split support
    rrobin  : integer := 0;-- round-robin arbitration
    timeout : integer range 0 to 255 := 0;  -- HREADY timeout
    ioaddr  : ahb_addr_type := 16#fff#;  -- I/O area MSB address
    iomask  : ahb_addr_type := 16#fff#;   -- I/O area address mask
    cfgaddr : ahb_addr_type := 16#ff0#;  -- config area MSB address
    cfgmask : ahb_addr_type := 16#ff0#; -- config area address maskk
    nahbm   : integer range 1 to NAHBMST := NAHBMST; -- number of masters
    nahbs   : integer range 1 to NAHBSLV := NAHBSLV; -- number of slaves
    ioen    : integer range 0 to 15 := 1;   -- enable I/O area
    disirq  : integer range 0 to 1 := 0; -- disable interrupt routing
    fixbrst : integer range 0 to 1 := 0; -- support fix-length bursts
    debug : integer range 0 to 2 := 2; -- print configuration to consolee
    fpnpen : integer range 0 to 1 := 0;     -- full PnP configuration decoding
    icheck  : integer range 0 to 1 := 1
    devid       : integer := 0;     -- unique device ID
    enbusmon    : integer range 0 to 1 := 0; --enable bus monitor
    assertwarn  : integer range 0 to 1 := 0; --enable assertions for warnings
    asserterr   : integer range 0 to 1 := 0; --enable assertions for errors
    hmstdisable : integer := 0; --disable master checks
    hslvdisable : integer := 0; --disable slave checks
    arbdisable  : integer := 0; --disable arbiter checks
    mprio       : integer := 0; --master with highest priority
    enebterm    : integer range 0 to 1 := 0  --enable early burst termination
);
  port (
    rst     : in  std_ulogic;
    clk     : in  std_ulogic;
    msti    : out ahb_mst_in_type;
    msto    : in  ahb_mst_out_vector;
    slvi    : out ahb_slv_in_type;
```

```
    slvo    : in  ahb_slv_out_vector;
    testen  : in  std_ulogic := '0';
    testrst : in  std_ulogic := '1';
    scanen  : in  std_ulogic := '0';
    testoen : in  std_ulogic := '1'
  );
  end component;
```

4.11 Instantiation

This example shows the core can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;

.
.

  -- AMBA signals
  signal ahbsi : ahb_slv_in_type;
  signal ahbso : ahb_slv_out_vector := (others => ahbs_none);
  signal ahbmi : ahb_mst_in_type;
  signal ahbmo : ahb_mst_out_vector := (others => ahbm_none);

begin

-- ARBITER

ahb0 : ahbctrl -- AHB arbiter/multiplexer
  generic map (defmast => CFG_DEFMST, split => CFG_SPLIT,
rrobin => CFG_RROBIN, ioaddr => CFG_AHBIO, nahbm => 8, nahbs => 8)
  port map (rstn, clkm, ahbmi, ahbmo, ahbsi, ahbso);


-- AHB slave

sr0 : srctrl generic map (hindex => 3)
port map (rstn, clkm, ahbsi, ahbso(3), memi, memo, sdo3);

-- AHB master

e1 : eth_oc
      generic map (mstndx => 2, slvndx => 5, ioaddr => CFG_ETHIO, irq => 12, memtech =>
memtech)
      port map (rstn, clkm, ahbsi, ahbso(5), ahbmi  => ahbmi,
ahbmo => ahbmo(2), ethi1, etho1);
...
end;
```

4.12 Debug print-out

If the debug generic is set to 2, the plug&play information of all attached AHB units are printed to the console during the start of simulation. Reporting starts by scanning the master interface array from 0 to NAHBMST - 1 (defined in the grlib.amba package). It checks each entry in the array for a valid vendor-id (all nonzero ids are considered valid) and if one is found, it also retrieves the device-id. The descriptions for these ids are obtained from the GRLIB.DEVICES package, and are then printed on standard out together with the master number. If the index check is enabled (done with a VHDL generic), the report module also checks if the hindex number returned in the record matches the array number of the record currently checked (the array index). If they do not match, the simulation is aborted and an error message is printed.

This procedure is repeated for slave interfaces found in the slave interface array. It is scanned from 0 to NAHBSLV - 1 and the same information is printed and the same checks are done as for the master interfaces. In addition, the address range and memory type is checked and printed. The address information includes type, address, mask, cacheable and pre-fetchable fields. From this information, the report module calculates the start address of the device and the size of the range. The information finally printed is type, start address, size, cacheability and pre-fetchability. The address ranges currently defined are AHB memory, AHB I/O and APB I/O. APB I/O ranges are ignored by this module.

```
# vsim -c -quiet leon3mp
VSIM 1> run
# LEON3 MP Demonstration design
# GRLIB Version 1.0.7
# Target technology: inferred,  memory library: inferred
# ahbctrl: AHB arbiter/multiplexer rev 1
# ahbctrl: Common I/O area disabled
# ahbctrl: Configuration area at 0xfffff000, 4 kbyte
# ahbctrl: mst0: Gaisler Research        Leon3 SPARC V8 Processor
# ahbctrl: mst1: Gaisler Research        AHB Debug UART
# ahbctrl: slv0: European Space Agency   Leon2 Memory Controller
# ahbctrl:       memory at 0x00000000, size 512 Mbyte, cacheable, prefetch
# ahbctrl:       memory at 0x20000000, size 512 Mbyte
# ahbctrl:       memory at 0x40000000, size 1024 Mbyte, cacheable, prefetch
# ahbctrl: slv1: Gaisler Research        AHB/APB Bridge
# ahbctrl:       memory at 0x80000000, size 1 Mbyte
# apbctrl: APB Bridge at 0x80000000 rev 1
# apbctrl: slv0: European Space Agency   Leon2 Memory Controller
# apbctrl:       I/O ports at 0x80000000, size 256 byte
# apbctrl: slv1: Gaisler Research        Generic UART
# apbctrl:       I/O ports at 0x80000100, size 256 byte
# apbctrl: slv2: Gaisler Research        Multi-processor Interrupt Ctrl.
# apbctrl:       I/O ports at 0x80000200, size 256 byte
# apbctrl: slv3: Gaisler Research        Modular Timer Unit
# apbctrl:       I/O ports at 0x80000300, size 256 byte
# apbctrl: slv7: Gaisler Research        AHB Debug UART
# apbctrl:       I/O ports at 0x80000700, size 256 byte
# apbctrl: slv11: Gaisler Research        General Purpose I/O port
# apbctrl:       I/O ports at 0x80000b00, size 256 byte
# grgpio11: 8-bit GPIO Unit rev 0
# gptimer3: GR Timer Unit rev 0, 8-bit scaler, 2 32-bit timers, irq 8
# irqmp: Multi-processor Interrupt Controller rev 3, #cpu 1
# apbuart1: Generic UART rev 1, fifo 4, irq 2
# ahbuart7: AHB Debug UART rev 0
# leon3_0: LEON3 SPARC V8 processor rev 0
# leon3_0: icache 1*8 kbyte, dcache 1*8 kbyte
VSIM 2>
```

Beilage 5

APBCTRL GRLIB Auszug

14 APBCTRL - AMBA AHB/APB bridge with plug&play support

14.1 Overview

The AMBA AHB/APB bridge is a APB bus master according the AMBA 2.0 standard.

The controller supports up to 16 slaves. The actual maximum number of slaves is defined in the GRLIB.AMBA package, in the VHDL constant NAPBSLV. The number of slaves can also be set using the *nslaves* VHDL generic.

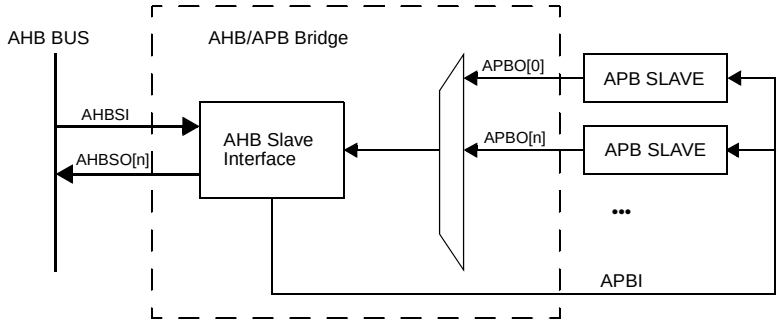


Figure 20. AHB/APB bridge block diagram

14.2 Operation

14.2.1 Decoding

Decoding (generation of PSEL) of APB slaves is done using the plug&play method explained in the GRLIB IP Library User's Manual. A slave can occupy any binary aligned address space with a size of 256 bytes - 1 Mbyte. Write to unassingned areas will be ignored, while reads from unassigned areas will return an arbitrary value. AHB error response will never be generated.

14.2.2 Plug&play information

GRLIB APB slaves contain two plug&play information words which are included in the APB records they drive on the bus (see the GRLIB IP Library User's Manual for more information). These records are combined into an array which is connected to the APB bridge.

The plug&play information is mapped on a read-only address area at the top 4 kbytes of the bridge address space. Each plug&play block occupies 8 bytes. The address of the plug&play information for a certain unit is defined by its bus index. If the bridge is mapped on AHB address 0x80000000, the address for the plug&play records is thus 0x800FF000 + n*8.

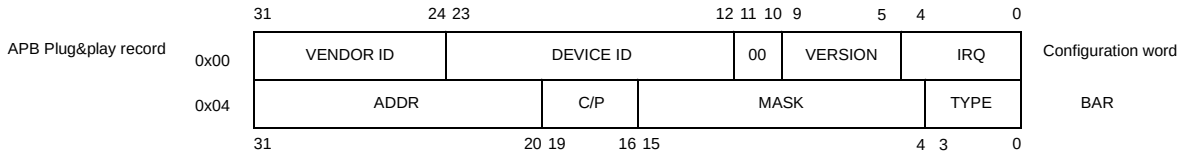


Figure 21. APB plug&play information

14.3 APB bus monitor

An APB bus monitor is integrated into the core. It is enabled with the enbusmon generic. It has the same functionality as the APB parts in the AMBA monitor core (AMBAMON). For more information on which rules are checked se the AMBAMON documentation.

14.4 Vendor and device identifiers

The core has vendor identifier 0x01 (Gaisler Research) and device identifier 0x006. For description of vendor and device identifiers see GRLIB IP Library User's Manual.

14.5 Configuration options

Table 67 shows the configuration options of the core (VHDL generics).

Table 67. Configuration options

Generic	Function	Allowed range	Default
hindex	AHB slave index	0 - NAHBSLV-1	0
haddr	The MSB address of the AHB area. Sets the 12 most significant bits in the 32-bit AHB address.	0 - 16#FFF#	16#FFF#
hmask	The AHB area address mask. Sets the size of the AHB area and the start address together with haddr.	0 - 16#FFF#	16#FFF#
nslaves	The maximum number of slaves	1 - NAPBSLV	NAPBSLV
debug	Print debug information during simulation	0 - 2	2
icheck	Enable bus index checking (PINDEX)	0 - 1	1
enbusmon	Enable APB bus monitor	0 - 1	0
asserterr	Enable assertions for AMBA requirements. Violations are asserted with severity error.	0 - 1	0
assertwarn	Enable assertions for AMBA recommendations. Violations are asserted with severity warning.	0 - 1	0
pslvdisable	Disable APB slave rule check. To disable a slave rule check a value is assigned so that the binary representation contains a one at the position corresponding to the rule number, e.g 0x80 disables rule 7.	N/A	0

14.6 Signal descriptions

Table 68 shows the interface signals of the core (VHDL ports).

Table 68. Signal descriptions

Signal name	Field	Type	Function	Active
RST	N/A	Input	AHB reset	Low
CLK	N/A	Input	AHB clock	-
AHBI	*	Input	AHB slave input	-
AHBO	*	Output	AHB slave output	-
APBI	*	Output	APB slave inputs	-
APBO	*	Input	APB slave outputs	-

* see GRLIB IP Library User's Manual

14.7 Library dependencies

Table 69 shows libraries used when instantiating the core (VHDL libraries).

Table 69. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Types	AMBA signal type definitions

14.8 Component declaration

```
library grlib;
use grlib.amba.all;

component apbctrl
  generic (
    hindex  : integer := 0;
    haddr   : integer := 0;
    hmask   : integer := 16#fff#;
    nslaves : integer range 1 to NAPBSLV := NAPBSLV;
    debug   : integer range 0 to 2 := 2;   -- print config to console
    icheck  : integer range 0 to 1 := 1
  );
  port (
    rst     : in  std_ulogic;
    clk     : in  std_ulogic;
    ahbi    : in  ahb_slv_in_type;
    ahbo    : out ahb_slv_out_type;
    apbi    : out apb_slv_in_type;
    apbo    : in  apb_slv_out_vector
  );
  end component;
```

14.9 Instantiation

This example shows how an APB bridge can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
use work.debug.all;

.
.

  -- AMBA signals

  signal ahbsi : ahb_slv_in_type;
  signal ahbso : ahb_slv_out_vector := (others => ahbs_none);

signal apbi  : apb_slv_in_type;
signal apbo  : apb_slv_out_vector := (others => apb_none);


begin


-- APB bridge

apb0 : apbctrl-- AHB/APB bridge
  generic map (hindex => 1, haddr => CFG_APBADDR)
  port map (rstn, clk, ahbsi, ahbso(1), apbi, apbo );
```

```
-- APB slaves

uart1 : apbuart
    generic map (pindex => 1, paddr => 1,  pirq => 2)
    port map (rstn, clk, apbi, apbo(1), u1i, u1o);

irqctrl0 : irqmp
    generic map (pindex => 2, paddr => 2)
    port map (rstn, clk, apbi, apbo(2), irqo, irqi);


...
end;
```

14.10 Debug print-out

The APB bridge can print-out the plug-play information from the attached during simulation. This is enabled by setting the debug VHDL generic to 2. Reporting starts by scanning the array from 0 to NAPBSLV - 1 (defined in the grlib.amba package). It checks each entry in the array for a valid vendor-id (all nonzero ids are considered valid) and if one is found, it also retrieves the device-id. The description for these ids are obtained from the GRLIB.DEVICES package, and is printed on standard out together with the slave number. If the index check is enabled (done with a VHDL generic), the report module also checks if the pindex number returned in the record matches the array number of the record currently checked (the array index). If they do not match, the simulation is aborted and an error message is printed.

The address range and memory type is also checked and printed. The address information includes type, address and mask. The address ranges currently defined are AHB memory, AHB I/O and APB I/O. All APB devices are in the APB I/O range so the type does not have to be checked. From this information, the report module calculates the start address of the device and the size of the range. The information finally printed is start address and size.

Beilage 6

DSU3 GRLIB Auszug

29 DSU3 - LEON3 Hardware Debug Support Unit

29.1 Overview

To simplify debugging on target hardware, the LEON3 processor implements a debug mode during which the pipeline is idle and the processor is controlled through a special debug interface. The LEON3 Debug Support Unit (DSU) is used to control the processor during debug mode. The DSU acts as an AHB slave and can be accessed by any AHB master. An external debug host can therefore access the DSU through several different interfaces. Such an interface can be a serial UART (RS232), JTAG, PCI, USB or ethernet. The DSU supports multi-processor systems and can handle up to 16 processors.

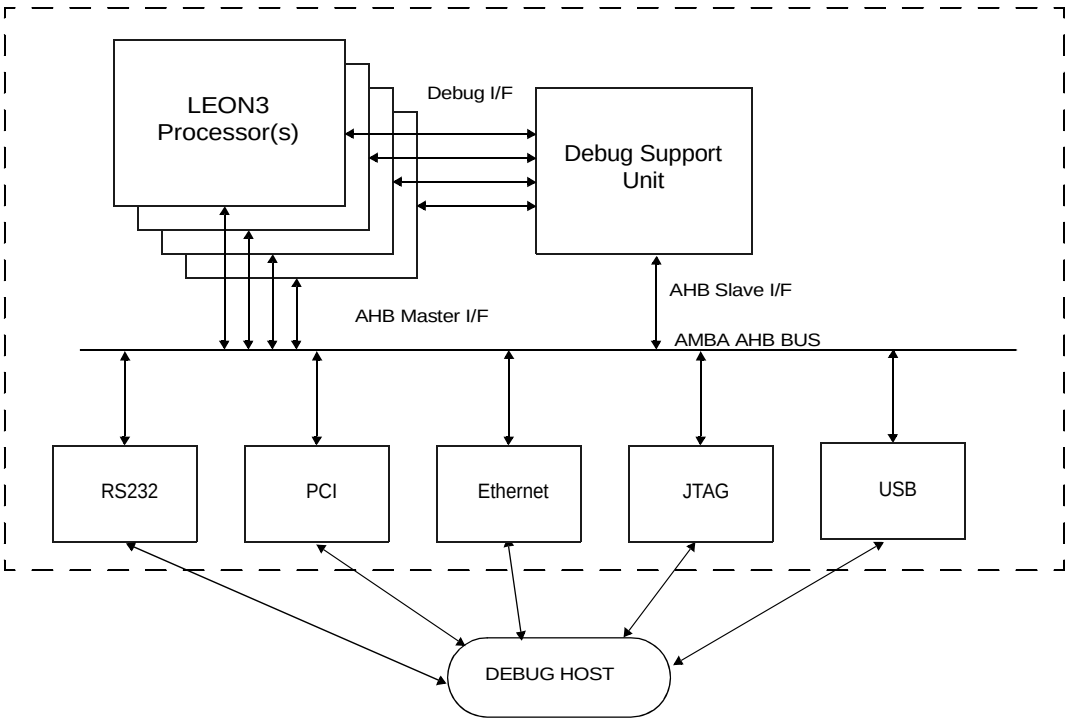


Figure 78. LEON3/DSU Connection

29.2 Operation

Through the DSU AHB slave interface, any AHB master can access the processor registers and the contents of the instruction trace buffer. The DSU control registers can be accessed at any time, while the processor registers, caches and trace buffer can only be accessed when the processor has entered debug mode. In debug mode, the processor pipeline is held and the processor state can be accessed by the DSU. Entering the debug mode can occur on the following events:

- executing a breakpoint instruction (ta 1)
- integer unit hardware breakpoint/watchpoint hit (trap 0xb)
- rising edge of the external break signal (DSUBRE)
- setting the break-now (BN) bit in the DSU control register
- a trap that would cause the processor to enter error mode
- occurrence of any, or a selection of traps as defined in the DSU control register
- after a single-step operation
- one of the processors in a multiprocessor system has entered the debug mode

DSU AHB breakpoint hit

The debug mode can only be entered when the debug support unit is enabled through an external signal (DSUEN). When the debug mode is entered, the following actions are taken:

- PC and nPC are saved in temporary registers (accessible by the debug unit)
- an output signal (DSUACT) is asserted to indicate the debug state
- the timer unit is (optionally) stopped to freeze the LEON timers and watchdog

The instruction that caused the processor to enter debug mode is not executed, and the processor state is kept unmodified. Execution is resumed by clearing the BN bit in the DSU control register or by deasserting DSUEN. The timer unit will be re-enabled and execution will continue from the saved PC and nPC. Debug mode can also be entered after the processor has entered error mode, for instance when an application has terminated and halted the processor. The error mode can be reset and the processor restarted at any address.

When a processor is in the debug mode, an access to ASI diagnostic area is forwarded to the IU which performs access with ASI equal to value in the DSU ASI register and address consisting of 20 LSB bits of the original address.

29.3 AHB Trace Buffer

The AHB trace buffer consists of a circular buffer that stores AHB data transfers. The address, data and various control signals of the AHB bus are stored and can be read out for later analysis. The trace buffer is 128 bits wide, the information stored is indicated in the table below:

*Table 209.*AHB Trace buffer data allocation

Bits	Name	Definition
127	AHB breakpoint hit	Set to ‘1’ if a DSU AHB breakpoint hit occurred.
126	-	Not used
125:96	Time tag	DSU time tag counter
95	-	Not used
94:80	Hirq	AHB HIRQ[15:1]
79	Hwrite	AHB HWRITE
78:77	Htrans	AHB HTRANS
76:74	Hsize	AHB HSIZE
73:71	Hburst	AHB HBURST
70:67	Hmaster	AHB HMASTER
66	Hmastlock	AHB HMASTLOCK
65:64	Hresp	AHB HRESP
63:32	Load/Store data	AHB HRDATA or HWDATA
31:0	Load/Store address	AHB HADDR

In addition to the AHB signals, the DSU time tag counter is also stored in the trace.

The trace buffer is enabled by setting the enable bit (EN) in the trace control register. Each AHB transfer is then stored in the buffer in a circular manner. The address to which the next transfer is written is held in the trace buffer index register, and is automatically incremented after each transfer. Tracing is stopped when the EN bit is reset, or when a AHB breakpoint is hit. Tracing is temporarily suspended when the processor enters debug mode. Note that neither the trace buffer memory nor the breakpoint registers (see below) can be read/written by software when the trace buffer is enabled.

29.4 Instruction trace buffer

The instruction trace buffer consists of a circular buffer that stores executed instructions. The instruction trace buffer is located in the processor, and read out via the DSU. The trace buffer is 128 bits wide, the information stored is indicated in the table below:

*Table 210.*Instruction trace buffer data allocation

Bits	Name	Definition
127	-	Unused
126	Multi-cycle instruction	Set to ‘1’ on the second and third instance of a multi-cycle instruction (LDD, ST or FPOP)
125:96	Time tag	The value of the DSU time tag counter
95:64	Load/Store parameters	Instruction result, Store address or Store data
63:34	Program counter	Program counter (2 lsb bits removed since they are always zero)
33	Instruction trap	Set to ‘1’ if traced instruction trapped
32	Processor error mode	Set to ‘1’ if the traced instruction caused processor error mode
31:0	Opcode	Instruction opcode

During tracing, one instruction is stored per line in the trace buffer with the exception of multi-cycle instructions. Multi-cycle instructions are entered two or three times in the trace buffer. For store instructions, bits [63:32] correspond to the store address on the first entry and to the stored data on the second entry (and third in case of STD). Bit 126 is set on the second and third entry to indicate this. A double load (LDD) is entered twice in the trace buffer, with bits [63:32] containing the loaded data. Multiply and divide instructions are entered twice, but only the last entry contains the result. Bit 126 is set for the second entry. For FPU operation producing a double-precision result, the first entry puts the MSB 32 bits of the results in bit [63:32] while the second entry puts the LSB 32 bits in this field.

When the processor enters debug mode, tracing is suspended. The trace buffer and the trace buffer control register can be read and written while the processor is in the debug mode. During the instruction tracing (processor in normal mode) the trace buffer and the trace buffer control register can not be accessed.

29.5 DSU memory map

The DSU memory map can be seen in table 211 below. In a multiprocessor systems, the register map is duplicated and address bits 27 - 24 are used to index the processor.

*Table 211.*DSU memory map

Address offset	Register
0x000000	DSU control register
0x000008	Time tag counter
0x000020	Break and Single Step register
0x000024	Debug Mode Mask register
0x000040	AHB trace buffer control register
0x000044	AHB trace buffer index register
0x000050	AHB breakpoint address 1
0x000054	AHB mask register 1
0x000058	AHB breakpoint address 2
0x00005c	AHB mask register 2
0x100000 - 0x10FFFF	Instruction trace buffer (..0: Trace bits 127 - 96, ..4: Trace bits 95 - 64, ..8: Trace bits 63 - 32, ..C : Trace bits 31 - 0)
0x110000	Instruction Trace buffer control register
0x200000 - 0x210000	AHB trace buffer (..0: Trace bits 127 - 96, ..4: Trace bits 95 - 64, ..8: Trace bits 63 - 32, ..C : Trace bits 31 - 0)
0x300000 - 0x3007FC	IU register file
0x300800 - 0x300FFC	IU register file check bits (LEON3FT only)
0x301000 - 0x30107C	FPU register file
0x400000 - 0x4FFFFC	IU special purpose registers
0x400000	Y register
0x400004	PSR register
0x400008	WIM register
0x40000C	TBR register
0x400010	PC register
0x400014	NPC register
0x400018	FSR register
0x40001C	CPSR register
0x400020	DSU trap register
0x400024	DSU ASI register
0x400040 - 0x40007C	ASR16 - ASR31 (when implemented)
0x700000 - 0x7FFFFC	ASI diagnostic access (ASI = value in DSU ASI register, address = address[19:0]) ASI = 0x9 : Local instruction RAM ASI = 0xB : Local data RAM ASI = 0xC : Instruction cache tags ASI = 0xD : Instruction cache data ASI = 0xE : Data cache tags ASI = 0xF : Data cache data ASI = 0x1E : Separate snoop tags

The addresses of the IU registers depends on how many register windows has been implemented:

- %o*n* : 0x300000 + (((psr.cwp * 64) + 32 + *n**4) mod (NWINDOWS*64))
- %l*n* : 0x300000 + (((psr.cwp * 64) + 64 + *n**4) mod (NWINDOWS*64))
- %i*n* : 0x300000 + (((psr.cwp * 64) + 96 + *n**4) mod (NWINDOWS*64))
- %g*n* : 0x300000 + (NWINDOWS*64)
- %f*n* : 0x301000 + *n**4

29.6 DSU registers

29.6.1 DSU control register

The DSU is controlled by the DSU control register:

.

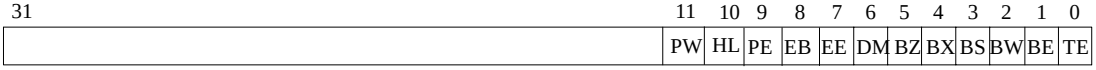


Figure 79. DSU control register

[0]: Trace enable (TE). Enables instruction tracing. If set the instructions will be stored in the trace buffer. Remains set when then processor enters debug or error mode.

[1]: Break on error (BE) - if set, will force the processor to debug mode when the processor would have entered error condition (trap in trap).

[2]: Break on IU watchpoint (BW)- if set, debug mode will be forced on a IU watchpoint (trap 0xb).

[3]: Break on S/W breakpoint (BS) - if set, debug mode will be forced when an breakpoint instruction (ta 1) is executed.

[4]: Break on trap (BX) - if set, will force the processor into debug mode when any trap occurs.

[5]: Break on error traps (BZ) - if set, will force the processor into debug mode on all *except* the following traps: priviledged_instruction, fpu_disabled, window_overflow, window_underflow, asynchronous_interrupt, ticc_trap.

[6]: Debug mode (DM). Indicates when the processor has entered debug mode (read-only).

[7]: EE - value of the external DSUEN signal (read-only)

[8]: EB - value of the external DSUBRE signal (read-only)

[9]: Processor error mode (PE) - returns ‘1’ on read when processor is in error mode, else ‘0’. If written with ‘1’, it will clear the error and halt mode.

[10]: Processor halt (HL). Returns ‘1’ on read when processor is halted. If the processor is in debug mode, setting this bit will put the processor in halt mode.

[11]: Power down (PW). Returns ‘1’ when processor in in power-down mode.

29.6.2 DSU Break and Single Step register

This register is used to break or single step the processor(s). This register controls all processors in a multi-processor system, and is only accessible in the DSU memory map of processor 0.

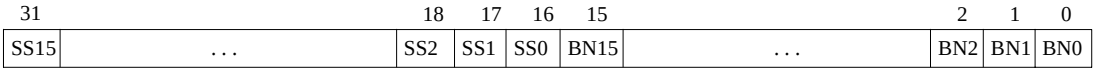


Figure 80. DSU Break and Single Step register

[15:0] : Break now (BNx) -Force processor x into debug mode if the Break on watchpoint (BW) bit in the processors DSU control register is set. If cleared, the processor x will resume execution.

[31:16] : Single step (SSx) - if set, the processor x will execute one instruction and return to debug mode. The bit remains set after the processor goes into the debug mode.

29.6.3 DSU Debug Mode Mask Register

When one of the processors in a multiprocessor LEON3 system enters the debug mode the value of the DSU Debug Mode Mask register determines if the other processors are forced in the debug mode. This register controls all processors in a multi-processor system, and is only accessible in the DSU memory map of processor 0.

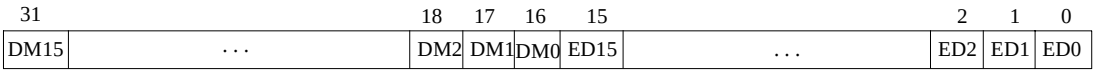


Figure 81. DSU Debug Mode Mask register

[15:0] : Enter debug mode (EDx) - Force processor x into debug mode if any of processors in a multiprocessor system enters the debug mode. If 0, the processor x will not enter the debug mode.

[31:16]: Debug mode mask. If set, the corresponding processor will not be able to force running processors into debug mode even if it enters debug mode.

29.6.4 DSU trap register

The DSU trap register is a read-only register that indicates which SPARC trap type that caused the processor to enter debug mode. When debug mode is force by setting the BN bit in the DSU control register, the trap type will be 0xb (hardware watchpoint trap).

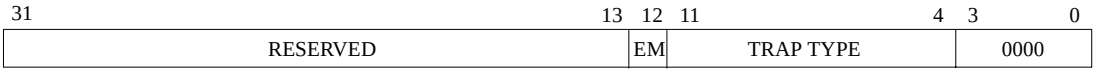


Figure 82. DSU trap register

[11:4]: 8-bit SPARC trap type

[12]: Error mode (EM). Set if the trap would have cause the processor to enter error mode.

29.6.5 Trace buffer time tag counter

The trace buffer time tag counter is incremented each clock as long as the processor is running. The counter is stopped when the processor enters debug mode, and restarted when execution is resumed.

Figure 83. Trace buffer time tag counter

The value is used as time tag in the instruction and AHB trace buffer.

The width of the timer (up to 30 bits) is configurable through the DSU generic port.

29.6.6 DSU ASI register

The DSU can perform diagnostic accesses to different ASI areas. The value in the ASI diagnostic access register is used as ASI while the address is supplied from the DSU.

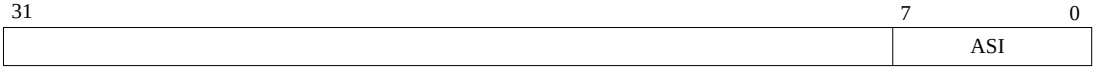


Figure 84. ASI diagnostic access register

[7:0]: ASI to be used on diagnostic ASI access

29.6.7 AHB Trace buffer control register

The AHB trace buffer is controlled by the AHB trace buffer control register:

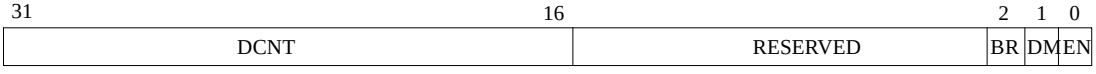


Figure 85. AHB trace buffer control register

[0]: Trace enable (EN). Enables the trace buffer.

[1]: Delay counter mode (DM). Indicates that the trace buffer is in delay counter mode.

[2]: Break (BR). If set, the processor will be put in debug mode when AHB trace buffer stops due to AHB breakpoint hit.

[31:16] Trace buffer delay counter (DCNT). Note that the number of bits actually implemented depends on the size of the trace buffer.

29.6.8 AHB trace buffer index register

The AHB trace buffer index register contains the address of the next trace line to be written.

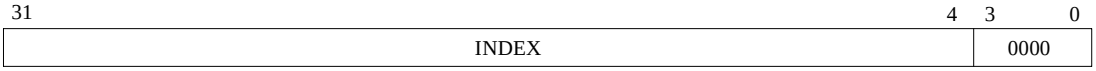


Figure 86. AHB trace buffer index register

31:4 Trace buffer index counter (INDEX). Note that the number of bits actually implemented depends on the size of the trace buffer.

29.6.9 AHB trace buffer breakpoint registers

The DSU contains two breakpoint registers for matching AHB addresses. A breakpoint hit is used to freeze the trace buffer by automatically clearing the enable bit. Freezing can be delayed by programming the DCNT field in the trace buffer control register to a non-zero value. In this case, the DCNT value will be decremented for each additional trace until it reaches zero, after which the trace buffer is frozen. A mask register is associated with each breakpoint, allowing breaking on a block of addresses. Only address bits with the corresponding mask bit set to ‘1’ are compared during breakpoint detection. To break on AHB load or store accesses, the LD and/or ST bits should be set.

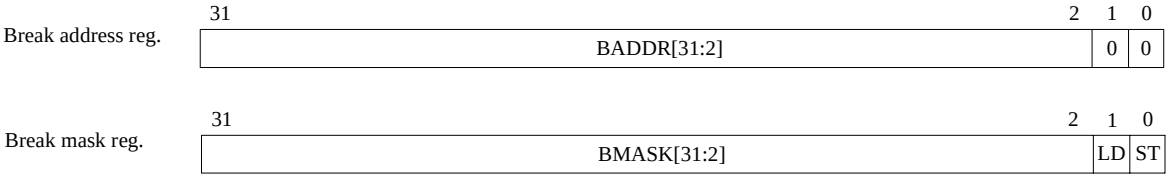


Figure 87. Trace buffer breakpoint registers

[31:2]: Breakpoint address (bits 31:2)
[31:2]: Breakpoint mask (see text)
[1]: LD - break on data load address
[0]: ST - beak on data store address

29.6.10 Instruction trace control register

The instruction trace control register contains a pointer that indicates the next line of the instruction trace buffer to be written.

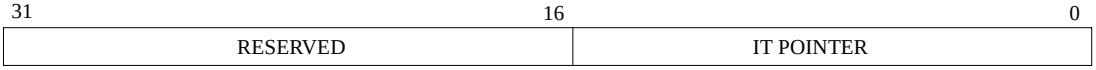


Figure 88. Instruction trace control register

[15:0] Instruction trace pointer. Note that the number of bits actually implemented depends on the size of the trace buffer.

29.7 Vendor and device identifiers

The core has vendor identifier 0x01 (Gaisler Research) and device identifier 0x017. For a description of vendor and device identifiers see GRLIB IP Library User’s Manual.

29.8 Technology mapping

DSU3 has one technology mapping generic, *tech*. This generic controls the implementation of which tehcnology will be used to implement the trace buffer memories. The AHB trace buffer will use four identical *syncram* block to implement the buffer memory. The all syncrams will be 32-bit wide. The depth will depend on the KBYTES generic, which indicates the total size of trace buffer in Kbytes. If KBYTES = 1 (1 Kbyte), then four RAM blocks of 64x32 will be used. If KBYTES = 2, then the RAM blocks will be 128x32 and so on.

29.9 Configuration options

Table 212 shows the configuration options of the core (VHDL generics).

*Table 212.*Configuration options

Generic	Function	Allowed range	Default
hindex	AHB slave index	0 - AHBSLVMAX-1	0
haddr	AHB slave address (AHB[31:20])	0 - 16#FFF#	16#900#
hmask	AHB slave address mask	0 - 16#FFF#	16#F00#
ncpu	Number of attached processors	1 - 16	1
tbits	Number of bits in the time tag counter	2 - 30	30
tech	Memory technology for trace buffer RAM	0 - TECHMAX-1	0 (inferred)
kbytes	Size of trace buffer memory in Kbytes. A value of 0 will disable the trace buffer function.	0 - 64	0 (disabled)

29.10 Signal descriptions

Table 213 shows the interface signals of the core (VHDL ports).

*Table 213.*Signal descriptions

Signal name	Field	Type	Function	Active
RST	N/A	Input	Reset	Low
CLK	N/A	Input	Clock	-
AHBMI	*	Input	AHB master input signals	-
AHBSI	*	Input	AHB slave input signals	-
AHBSO	*	Output	AHB slave output signals	-
DBGI	-	Input	Debug signals from LEON3	-
DBGO	-	Output	Debug signals to LEON3	-
DSUI	ENABLE	Input	DSU enable	High
	BREAK	Input	DSU break	High
DSUO	ACTIVE	Output	Debug mode	High
	PWD[n-1 : 0]	Output	Clock gating enable for processor [n]	High

* see GRLIB IP Library User’s Manual

29.11 Library dependencies

Table 214 shows libraries used when instantiating the core (VHDL libraries).

*Table 214.*Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AHB signal definitions
GAISLER	LEON3	Component, signals	Component declaration, signals declaration

29.12 Component declaration

The core has the following component declaration.

```
component dsu3
  generic (
    hindex : integer := 0;
    haddr : integer := 16#900#;
    hmask : integer := 16#f00#;
    ncpu    : integer := 1;
    tbits   : integer := 30;
    tech    : integer := 0;
    irq     : integer := 0;
    kbytes  : integer := 0
  );
  port (
    rst    : in  std_ulogic;
    clk    : in  std_ulogic;
    ahbmi  : in  ahb_mst_in_type;
    ahbsi  : in  ahb_slv_in_type;
    ahbso  : out ahb_slv_out_type;
    dbgi   : in l3_debug_out_vector(0 to NCPU-1);
    dbgo   : out l3_debug_in_vector(0 to NCPU-1);
    dsui   : in dsu_in_type;
    dsuo   : out dsu_out_type
  );
  end component;
```

29.13 Instantiation

This example shows how the core can be instantiated.

The DSU is always instantiated with at least one LEON3 processor. It is suitable to use a generate loop for the instantiation of the processors and DSU and showed below.

```
library ieee;
use ieee.std_logic_1164.all;
library grlib;
use grlib.amba.all;
library gaisler;
use gaisler.leon3.all;

constant NCPU : integer := 1; -- select number of processors

signal leon3i : l3_in_vector(0 to NCPU-1);
signal leon3o : l3_out_vector(0 to NCPU-1);
signal irqi   : irq_in_vector(0 to NCPU-1);
signal irqo   : irq_out_vector(0 to NCPU-1);

signal dbgi : l3_debug_in_vector(0 to NCPU-1);
signal dbgo : l3_debug_out_vector(0 to NCPU-1);

signal dsui   : dsu_in_type;
```

```
signal dsuo   : dsu_out_type;

.
begin

cpu : for i in 0 to NCPU-1 generate
    u0 : leon3s-- LEON3 processor
    generic map (ahbndx => i, fabtech => FABTECH, memtech => MEMTECH)
    port map (clkm, rstn, ahbmi, ahbmo(i), ahbsi, ahbsi, ahbso,
 irqi(i), irqo(i), dbgi(i), dbgo(i));
    irqi(i) <= leon3o(i).irq; leon3i(i).irq <= irqo(i);
end generate;

dsu0 : dsu3-- LEON3 Debug Support Unit
    generic map (ahbndx => 2, ncpu => NCPU, tech => memtech, kbytes => 2)
    port map (rstn, clkm, ahbmi, ahbsi, ahbso(2), dbgo, dbgi, dsui, dsuo);
dsui.enable <= dsuen; dsui.break <= dsubre; dsuact <= dsuo.active;
```

Beilage 7

MCTRL GRLIB Auszug

58 MCTRL - Combined PROM/IO/SRAM/SDRAM Memory Controller

58.1 Overview

The memory controller handles a memory bus hosting PROM, memory mapped I/O devices, asynchronous static ram (SRAM) and synchronous dynamic ram (SDRAM). The controller acts as a slave on the AHB bus. The function of the memory controller is programmed through memory configuration registers 1, 2 & 3 (MCFG1, MCFG2 & MCFG3) through the APB bus. The memory bus supports four types of devices: prom, sram, sdram and local I/O. The memory bus can also be configured in 8- or 16-bit mode for applications with low memory and performance demands.

Chip-select decoding is done for two PROM banks, one I/O bank, five SRAM banks and two SDRAM banks.

The controller decodes three address spaces (PROM, I/O and RAM) whose mapping is determined through VHDL-generics.

Figure 195 shows how the connection to the different device types is made.

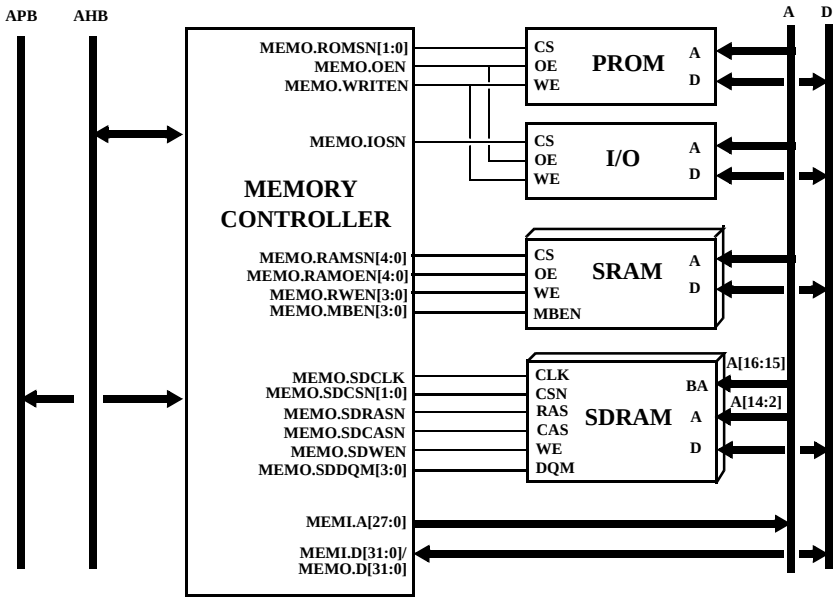


Figure 195. **Memory controller conected to AMBA bus and different types of memory devices**

58.2 PROM access

Accesses to prom have the same timing as RAM accesses, the differences being that PROM cycles can have up to 15 waitstates.

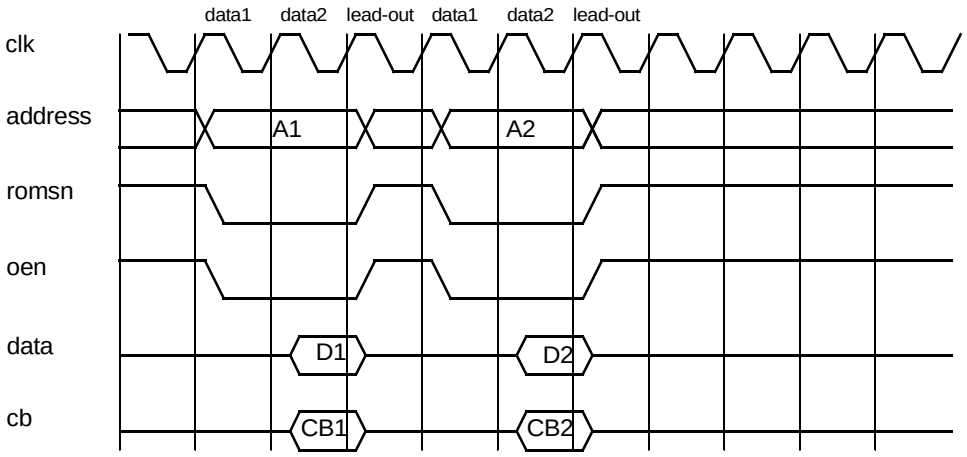


Figure 196. **Prom non-consecutive read cyclecs.**

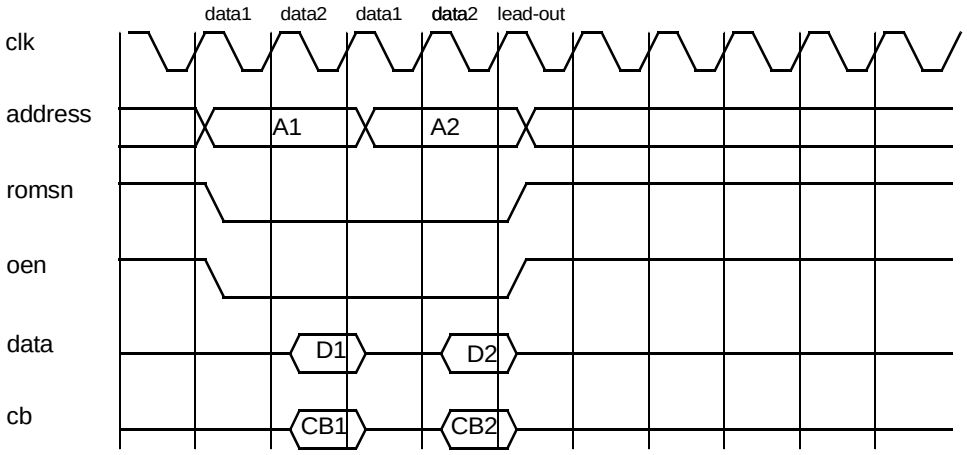


Figure 197. **Prom consecutive read cyclecs.**

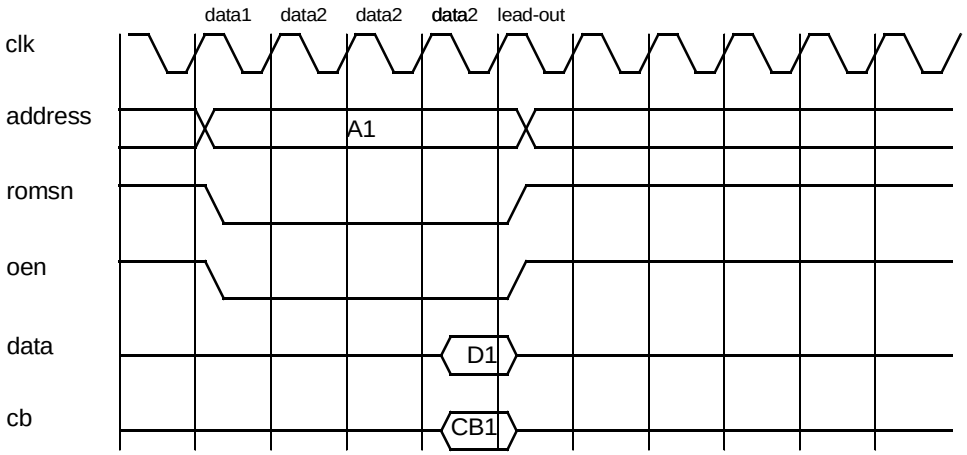


Figure 198. **Prom read access with two waitstates.**

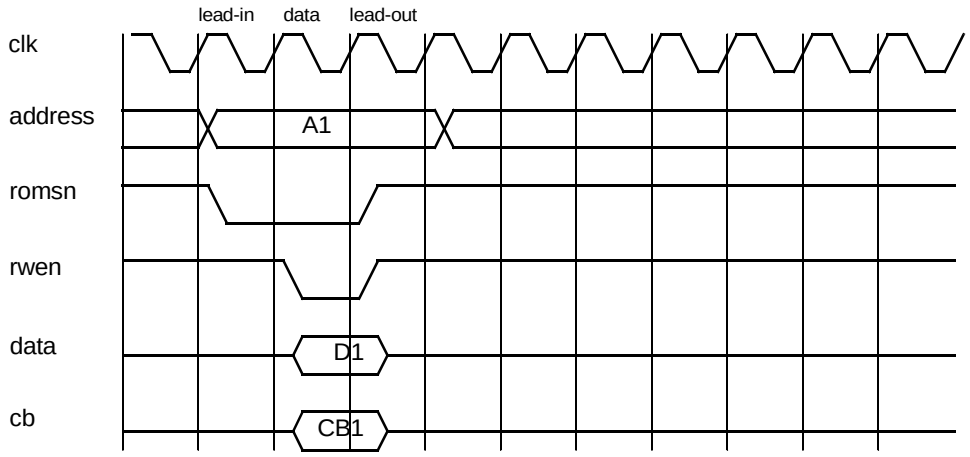


Figure 199. **Prom write cycle (0-waitstates)**

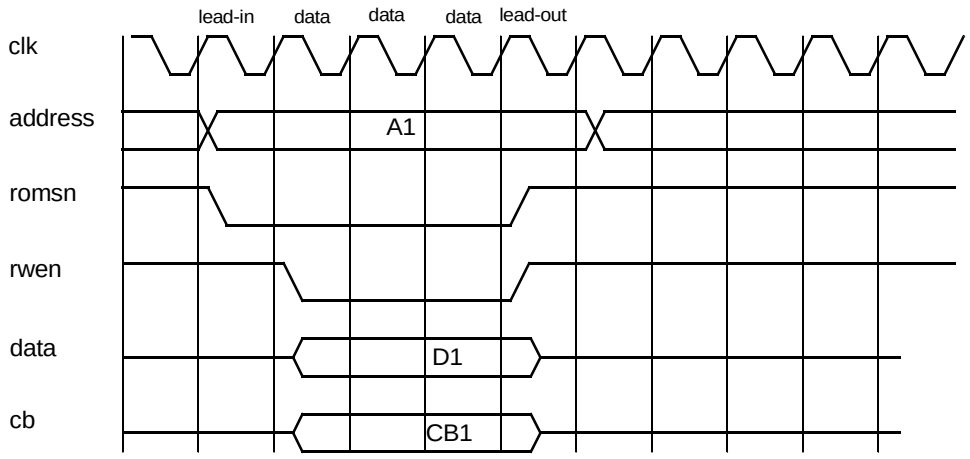


Figure 200. **Prom write cycle (2-waitstates)**

Two PROM chip-select signals are provided, MEMO.ROMSN[1:0]. MEMO.ROMSN[0] is asserted when the lower half of the PROM area as addressed while MEMO.ROMSN[1] is asserted for the upper half. When the VHDL model is configured to boot from internal prom, MEMO.ROMSN[0] is never asserted and all accesses to the lower half of the PROM area are mapped on the internal prom.

58.3 Memory mapped I/O

Accesses to I/O have similar timing to ROM/RAM accesses, the differences being that a additional waitstates can be inserted by de-asserting the MEMI.BRDYN signal. The I/O select signal (MEMO.IOSN) is delayed one clock to provide stable address before MEMO.IOSN is asserted.

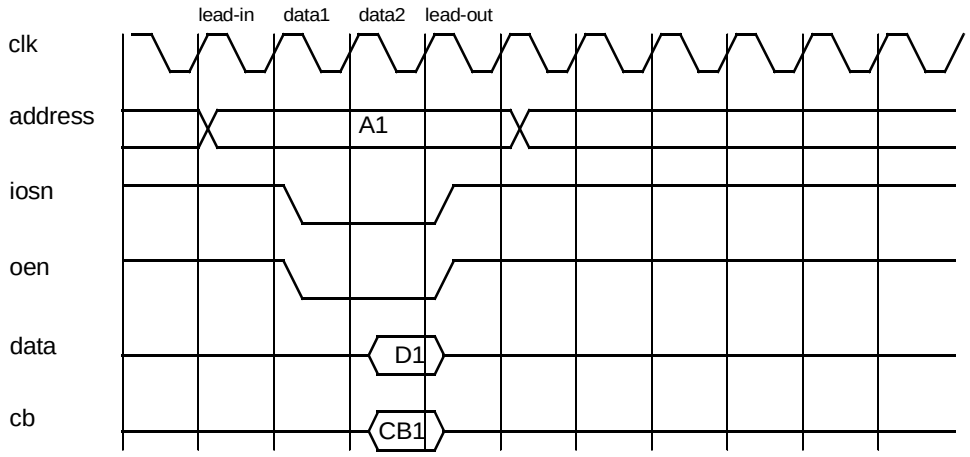


Figure 201. **I/O read cycle (0-waitstates)**

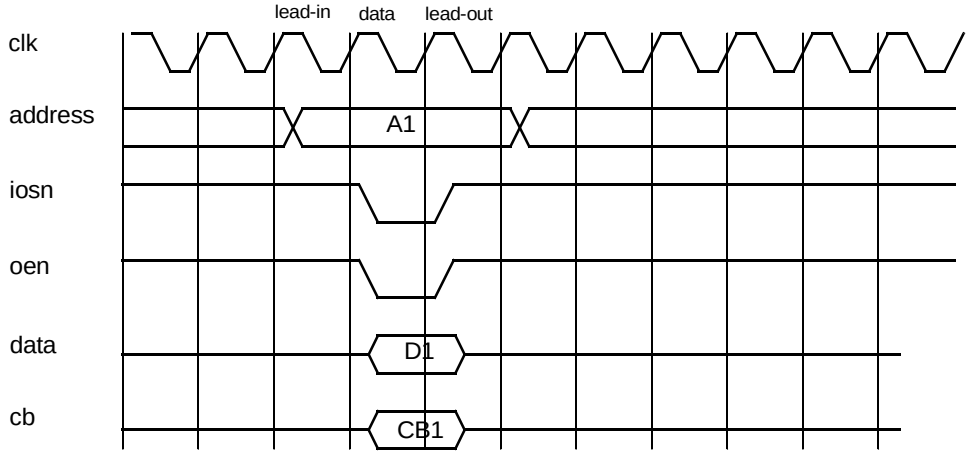


Figure 202. **I/O write cycle (0-waitstates)**

58.4 SRAM access

The SRAM area can be up to 1 Gbyte, divided on up to five RAM banks. The size of banks 1-4 (MEMO.RAMSN[3:0]is programmed in the RAM bank-size field (MCFG2[12:9]) and can be set in binary steps from 8 Kbyte to 256 Mbyte. The fifth bank (RAMSN[4]) decodes the upper 512 Mbyte (controlled by means of the *sdrasel* VHDL generic) and cannot be used simultaneously with SDRAM memory. A read access to SRAM consists of two data cycles and between zero and three waitstates. Accesses to MEMO.RAMSN[4] can further be stretched by de-asserting MEMI.BRDYN until the data is available. On non-consecutive accesses, a lead-out cycle is added after a read cycle to prevent bus contention due to slow turn-off time of memories or I/O devices. Figure 203 shows the basic read cycle waveform (zero waitstate).

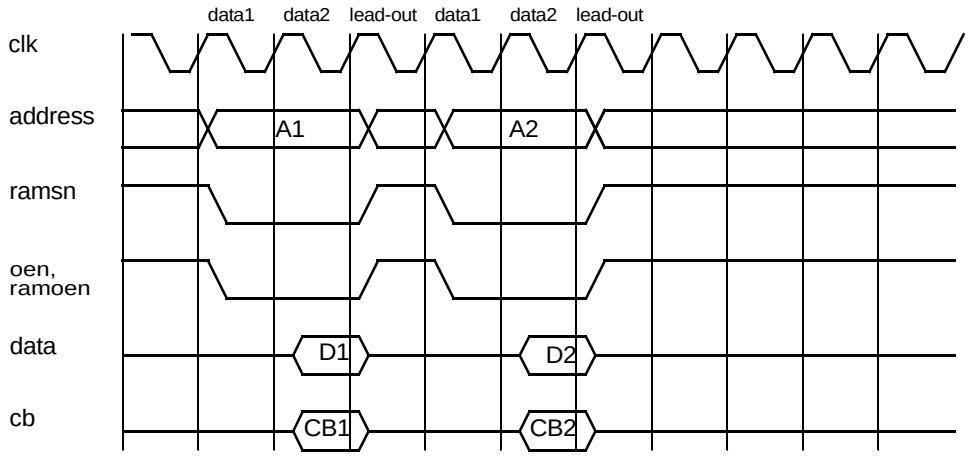


Figure 203. **SRAM non-consecutive read cyclecs.**

For read accesses to MEMO.RAMSN[4:0], a separate output enable signal (MEMO.RAMOEN[n]) is provided for each RAM bank and only asserted when that bank is selected. A write access is similar to the read access but takes a minimum of three cycles:

Through an (optional) feed-back loop from the write strobes, the data bus is guaranteed to be driven until the write strobes are de-asserted. Each byte lane has an individual write strobe to allow efficient byte and half-word writes. If the memory uses a common write strobe for the full 16- or 32-bit data, the read-modify-write bit in the MCFG2 register should be set to enable read-modify-write cycles for sub-word writes.

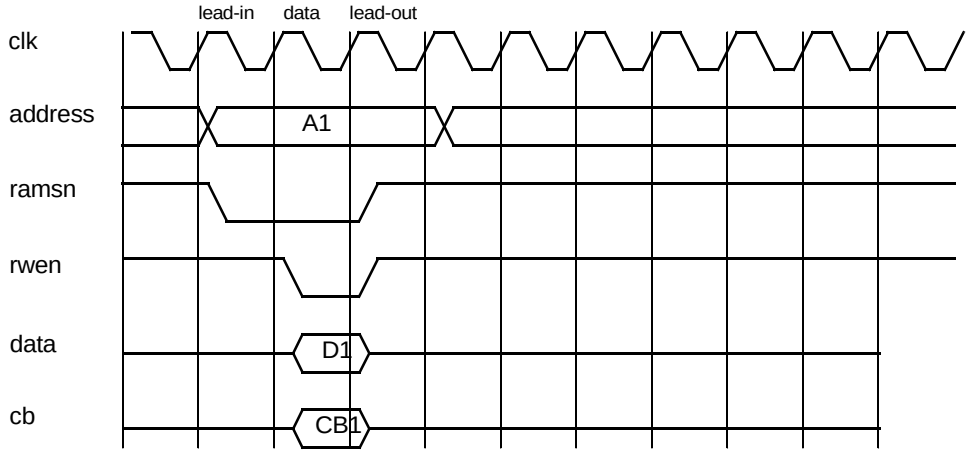


Figure 204. **Sram write cycle (0-waitstates)**

A drive signal vector for the data I/O-pads is provided which has one drive signal for each data bit. It can be used if the synthesis tool does not generate separate registers automatically for the current technology. This can remove timing problems with output delay.

58.5 8-bit and 16-bit PROM and SRAM access

To support applications with low memory and performance requirements efficiently, it is not necessary to always have full 32-bit memory banks. The SRAM and PROM areas can be individually configured for 8- or 16-bit operation by programming the ROM and RAM size fields in the memory configuration registers. Since read access to memory is always done on 32-bit word basis, read access to 8-bit memory will be transformed in a burst of four read cycles while access to 16-bit memory will

generate a burst of two 16-bits reads. During writes, only the necessary bytes will be writen. Figure 205 shows an interface example with 8-bit PROM and 8-bit SRAM. Figure 206 shows an example of a 16-bit memory interface.

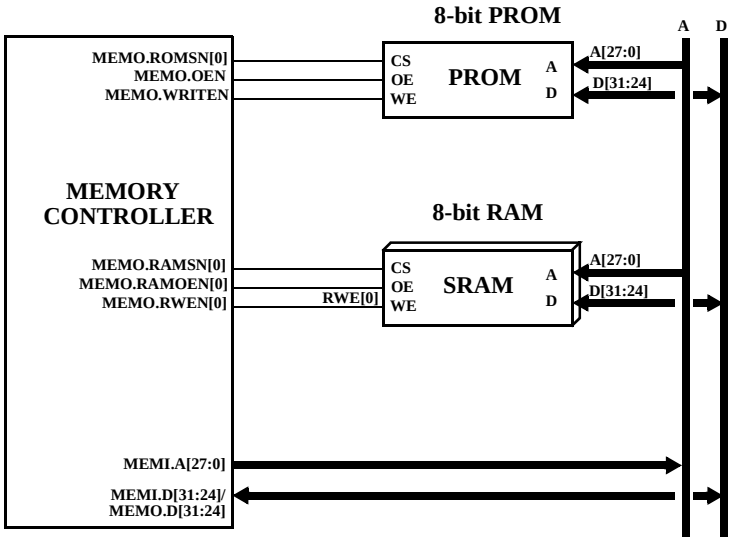


Figure 205. **8-bit memory interface example**

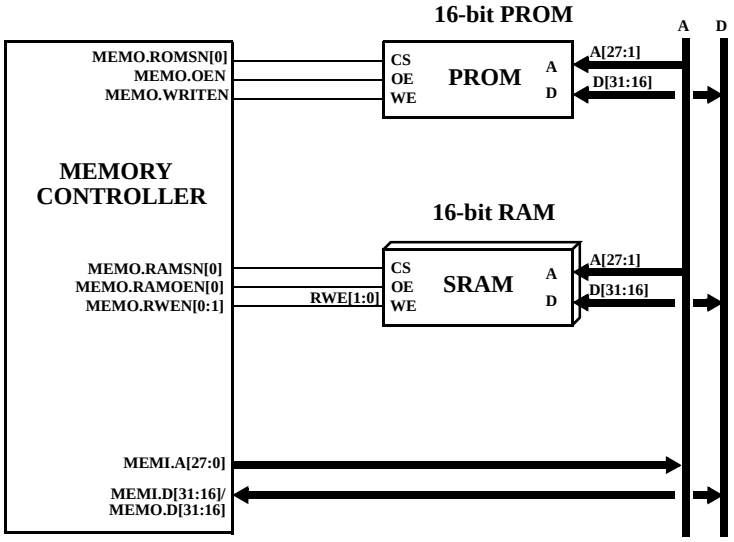


Figure 206. **16-bit memory interface example**

58.6 Burst cycles

To improve the bandwidth of the memory bus, accesses to consecutive addresses can be performed in burst mode. Burst transfers will be generated when the memory controller is accessed using an AHB burst request. These includes instruction cache-line fills, double loads and double stores. The timing of a burst cycle is identical to the programmed basic cycle with the exception that during read cycles, the lead-out cycle will only occurs after the last transfer.

58.7 8- and 16-bit I/O access

Similar to the PROM/RAM areas, the I/O area can also be configured to 8- or 16-bit mode. However, the I/O device will NOT be accessed by multiple 8/16 bit accesses as the memory areas, but only with one single access just as in 32-bit mode. To access an I/O device on a 16-bit bus, LDUH/STH instructions should be used while LDUB/STB should be used with an 8-bit bus.

58.8 SDRAM access

58.8.1 General

Synchronous dynamic RAM (SDRAM) access is supported to two banks of PC100/PC133 compatible devices. This is implemented by a special version of the SDCTRL SDRAM controller core from Gaisler Research, which is optionally instantiated as a sub-block. The SDRAM controller supports 64M, 256M and 512M devices with 8 - 12 column-address bits, and up to 13 row-address bits. The size of the two banks can be programmed in binary steps between 4 Mbyte and 512 Mbyte. The operation of the SDRAM controller is controlled through MCFG2 and MCFG3 (see below). Both 32- and 64-bit data bus width is supported, allowing the interface of 64-bit DIMM modules. The memory controller can be configured to use either a shared or separate bus connecting the controller and SDRAM devices. When the VHDL generic **mobile** is set to a value not equal to 0, the controller supports mobile SDRAM.

58.8.2 Address mapping

The two SDRAM chip-select signals are decoded. SDRAM area is mapped into the upper half of the RAM area defined by BAR2 register. When the SDRAM enable bit is set in MCFG2, the controller is enabled and mapped into upper half of the RAM area as long as the SRAM disable bit is not set. If the SRAM disable bit is set, all access to SRAM is disabled and the SDRAM banks are mapped into the lower half of the RAM area.

58.8.3 Initialisation

When the SDRAM controller is enabled, it automatically performs the SDRAM initialisation sequence of PRECHARGE, 2x AUTO-REFRESH and LOAD-MODE-REG on both banks simultaneously. When mobile SDRAM functionality is enabled the initialization sequence is appended by a LOAD-EXTMODE-REG command. The controller programs the SDRAM to use page burst on read and single location access on write.

58.8.4 Configurable SDRAM timing parameters

To provide optimum access cycles for different SDRAM devices (and at different frequencies), some SDRAM parameters can be programmed through memory configuration register 2 (MCFG2) The programmable SDRAM parameters can be seen in tabel 645.

***Table 645.*SDRAM programmable timing parameters**

Function	Parameter	Range	Unit
CAS latency, RAS/CAS delay	t_{CAS}, t_{RCD}	2 - 3	clocks
Precharge to activate	t_{RP}	2 - 3	clocks
Auto-refresh command period	t_{RFC}	3 - 11	clocks
Auto-refresh interval		10 - 32768	clocks

Remaining SDRAM timing parameters are according the PC100/PC133 specification.

When mobile SDRAM support is enabled, one additional timing parameter (TXSR) can be programmed though the Power-Saving configuration register.

*Table 646.*Mobile SDRAM programmable minimum timing parameters

SDRAM timing parameter	Minimum timing (clocks)
Exit Self Refresh mode to first valid command (t_{XSR})	tXSR

58.9 Refresh

The SDRAM controller contains a refresh function that periodically issues an AUTO-REFRESH command to both SDRAM banks. The period between the commands (in clock periods) is programmed in the refresh counter reload field in the MCFG3 register. Depending on SDRAM type, the required period is typically 7.8 or 15.6 µs (corresponding to 780 or 1560 clocks at 100 MHz). The generated refresh period is calculated as (reload value+1)/sysclk. The refresh function is enabled by setting bit 31 in MCFG2.

58.9.1 Self Refresh

The self refresh mode can be used to retain data in the SDRAM even when the rest of the system is powered down. When in the self refresh mode, the SDRAM retains data without external clocking and refresh are handled internally. The memory array that is refreshed during the self refresh operation is defined in the extended mode register. These settings can be changed by setting the PASR bits in the Power-Saving configuration register. The extended mode register is automatically updated when the PASR bits are changed. The supported “Partial Array Self Refresh” modes are: Full, Half, Quarter, Eighth, and Sixteenth array. “Partial Array Self Refresh” is only supported when mobile SDRAM functionality is enabled. To enable the self refresh mode, set the PMODE bits in the Power-Saving configuration register to “010” (Self Refresh). The controller will enter self refresh mode after every memory access (when the controller has been idle for 16 clock cycles), until the PMODE bits are cleared. When exiting this mode the controller introduce a delay defined by tXSR in the Power-Saving configuration register and a AUTO REFRESH command before any other memory access is allowed. The minimum duration of this mode is defined by tRAS. This mode is only available then the VHDL generic **mobile** >=1.

58.9.2 Power-Down

When entering the power-down mode all input and output buffers, excluding SDCKE, are deactivated. All data in the SDRAM is retained during this operation. To enable the power-down mode, set the PMODE bits in the Power-Saving configuration register to “001” (Power-Down). The controller will enter power-down mode after every memory access (when the controller has been idle for 16 clock cycles), until the PMODE bits is cleared. The REFRESH command will still be issued by the controller in this mode. When exiting this mode a delay of one clock cycles are added before issue any command to the memory. This mode is only available then the VHDL generic **mobile** >=1.

58.9.3 Deep Power-Down

The deep power-down operating mode is used to achieve maximum power reduction by eliminating the power of the memory array. Data will not be retained after the device enters deep power-down mode. To enable the deep power-down mode, set the PMODE bits in the Power-Saving configuration register to “101” (Deep Power-Down). To exit the deep power-down mode the PMODE bits in the Power-Saving configuration register must be cleared. The controller will respond with an AMBA ERROR response to an AMBA access, that will result in a memory access, during Deep Power-Down mode. This mode is only available then the VHDL generic **mobile** >=1 and mobile SDRAM functionality is enabled.

58.9.4 Temperature-Compensated Self Refresh

The settings for the temperature-compensation of the Self Refresh rate can be controlled by setting the TCSR bits in the Power-Saving configuration register. The extended mode register is automatically updated when the TCSR bits are changed. Note that some vendors implements a Internal Temperature-Compensated Self Refresh feature, which makes the memory to ignore the TCSR bits. This functionality is only available then the VHDL generic **mobile** >=1 and mobile SDRAM functionality is enabled.

58.9.5 Drive Strength

The drive strength of the output buffers can be controlled by setting the DS bits in the Power-Saving configuration register. The extended mode register is automatically updated when the DS bits are changed. The available options are: full, three-quarter, one-half, and one-quarter drive strengths. This functionality is only available then the VHDL generic **mobile** >=1 and mobile SDRAM functionality is enabled.

58.9.6 SDRAM commands

The controller can issue four SDRAM commands by writing to the SDRAM command field in MCFG2: PRE-CHARGE, AUTO-REFRESH, LOAD-MODE-REG (LMR) and LOAD-EXTMODE-REG (EMR). If the LMR command is issued, the CAS delay as programmed in MCFG2 will be used, remaining fields are fixed: page read burst, single location write, sequential burst. If the EMR command is issued, the DS, TCSR and PASR as programmed in Power-Saving configuration register will be used. To issue the EMR command, the EMR bit in the MCFG4 register has to be set. The command field will be cleared after a command has been executed. Note that when changing the value of the CAS delay, a LOAD-MODE-REGISTER command should be generated at the same time.

58.9.7 Read cycles

A read cycle is started by performing an ACTIVATE command to the desired bank and row, followed by a READ command after the programmed CAS delay. A read burst is performed if a burst access has been requested on the AHB bus. The read cycle is terminated with a PRE-CHARGE command, no banks are left open between two accesses.

58.9.8 Write cycles

Write cycles are performed similarly to read cycles, with the difference that WRITE commands are issued after activation. A write burst on the AHB bus will generate a burst of write commands without idle cycles in-between.

58.9.9 Address bus connection

The memory controller can be configured to either share the address and data buses with the SRAM, or to use separate address and data buses. When the buses are shared, the address bus of the SDRAMs should be connected to A[14:2], the bank address to A[16:15]. The MSB part of A[14:2] can be left unconnected if not used. When separate buses are used, the SDRAM address bus should be connected to SA[12:0] and the bank address to SA[14:13].

58.9.10 Data bus

SDRAM can be connected to the memory controller through the common or separate data bus. If the separate bus is used the width is configurable to 32 or 64 bits. 64-bit data bus allows the 64-bit SDRAM devices to be connected using the full data capacity of the devices. 64-bit SDRAM devices can be connected to 32-bit data bus if 64-bit data bus is not available but in this case only half the full data capacity will be used. There is a drive signal vector and separate data vector available for SDRAM. The drive vector has one drive signal for each data bit. These signals can be used to remove

timing problems with the output delay when a separate SDRAM bus is used. SDRAM bus signals are described in section 58.13, for configuration options refer to section 58.15.

58.9.11 Clocking

The SDRAM clock typically requires special synchronisation at layout level. For Xilinx and Altera device, the GR Clock Generator can be configured to produce a properly synchronised SDRAM clock. For other FPGA targets, the GR Clock Generator can produce an inverted clock.

58.10 Using bus ready signalling

The MEMI.BRDYN signal can be used to stretch access cycles to the I/O area and the ram area decoded by MEMO.RAMSN[4]. The accesses will always have at least the pre-programmed number of waitstates as defined in memory configuration registers 1 & 2, but will be further stretched until MEMI.BRDYN is asserted. MEMI.BRDYN should be asserted in the cycle preceding the last one. The use of MEMI.BRDYN can be enabled separately for the I/O and RAM areas.

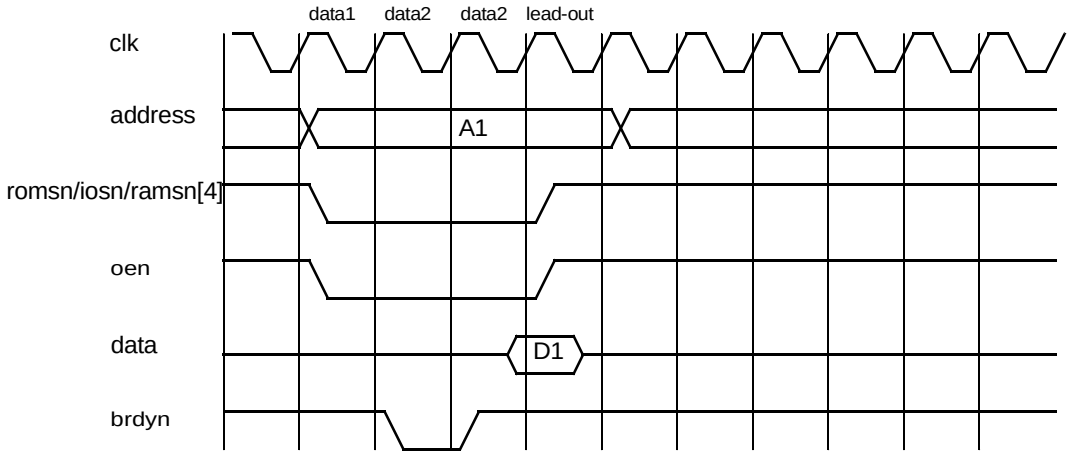


Figure 207. **READ cycle with one extra data2 cycle added with BRDYN (synchronous sampling). Lead-out cycle is only applicable for I/O accesses.**

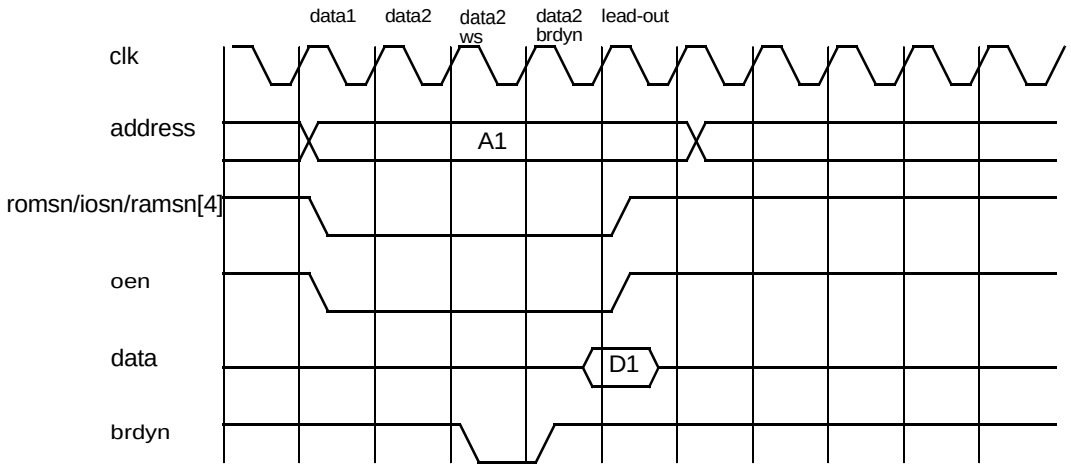


Figure 208. **Read cycle with one waitstate (configured) and one BRDYN generated waitstate (synchronous sampling).**

58.11 Access errors

An access error can be signalled by asserting the MEMI.BEXCN signal, which is sampled together with the data. If the usage of MEMI.BEXCN is enabled in memory configuration register 1, an error

response will be generated on the internal AMBA bus. MEMI.BEXCN can be enabled or disabled through memory configuration register 1, and is active for all areas (PROM, I/O an RAM).

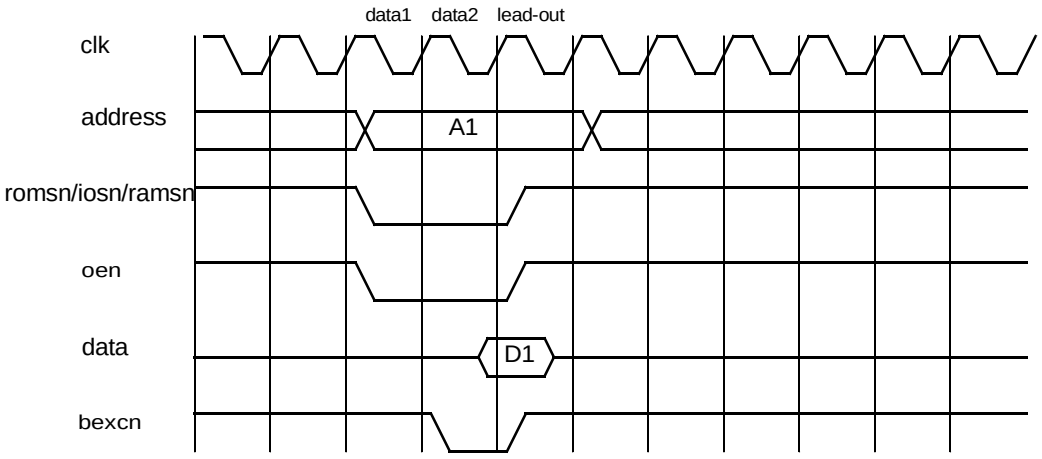


Figure 209. **Read cycle with BEXCN.**

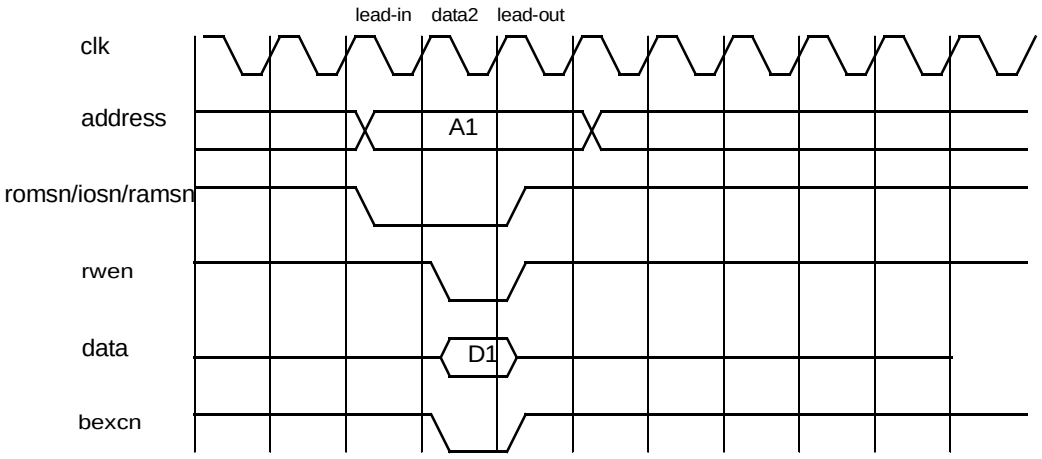


Figure 210. **Write cycle with BEXCN. Chip-select (iosn) is not asserted in lead-in cycle for io-accesses.**

58.12 Attaching an external DRAM controller

To attach an external DRAM controller, MEMO.RAMSN[4] should be used since it allows the cycle time to vary through the use of MEMI.BRDYN. In this way, delays can be inserted as required for opening of banks and refresh.

58.13 Registers

The memory controller is programmed through registers mapped into APB address space.

Table 647.**Memory controller registers**

APB address offset	Register
0x0	MCFG1
0x4	MCFG2
0x8	MCFG3
0xC	MCFG4 (Power-Saving configuration register)

58.13.1 Memory configuration register 1 (MCFG1)

Memory configuration register 1 is used to program the timing of rom and local I/O accesses.

Table 648. Memory configuration register 1.

31 29	28 27	26	25	24	23 20	19	18
RESERVED	IOBUSW	IBRDY	BEXCN		IO WAITSTATES	IOEN	

12	11	10	9 8	7 4	3 0
RESERVED	PWEN		PROM WIDTH	PROM WRITE WS	PROM READ WS

31 : 29	RESERVED
28 : 27	I/O bus width (IOBUSW) - Sets the data width of the I/O area (“00”=8, “01”=16, “10” =32).
26	I/O bus ready enable (IBRDY) - Enables bus ready (BRDYN) signalling for the I/O area. Reset to ‘0’.
25	Bus error enable (BEXCN) - Enables bus error signalling. Reset to ‘0’.
24	RESERVED
23 : 20	I/O waitstates (IO WAITSTATES) - Sets the number of waitstates during I/O accesses (“0000”=0, “0001”=1, “0010”=2,..., “1111”=15).
19	I/O enable (IOEN) - Enables accesses to the memory bus I/O area.
18:12	RESERVED
11	PROM write enable (PWEN) - Enables write cycles to the PROM area.
10	RESERVED
9 : 8	PROM width (PROM WIDTH) - Sets the data width of the PROM area (“00”=8, “01”=16, “10”=32).
7 : 4	PROM write waitstates (PROM WRITE WS) - Sets the number of wait states for PROM write cycles (“0000”=0, “0001”=1, “0010”=2,..., “1111”=15).
3 : 0	PROM read waitstates (PROM READ WS) - Sets the number of wait states for PROM read cycles (“0000”=0, “0001”=1, “0010”=2,...,”1111”=15). Reset to “1111”.

During power-up, the prom width (bits [9:8]) are set with value on MEMI.BWIDTH inputs. The prom waitstates fields are set to 15 (maximum). External bus error and bus ready are disabled. All other fields are undefined.

58.13.2 Memory configuration register 2 (MCFG2)

Memory configuration register 2 is used to control the timing of the SRAM and SDRAM.

Table 649. Memory configuration register 2.

31	30	29 28 27	26	25 24 23	22 21	20 19	18	17	16
SDRF	TRP	SDRAM TRFC	TCAS	SDRAM BANKSZ	SDRAM COLSZ	SDRAM CMD	D64	RES	MS

15	14	13	12 9	8	7	6	5 4	3 2	1 0
RES	SE	SI	RAM BANK SIZE		RBRDY	RMW	RAM WIDTH	RAM WRITE WS	RAM READ WS

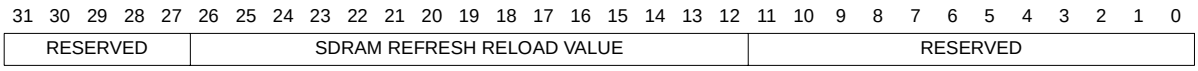
31	SDRAM refresh (SDRF) - Enables SDRAM refresh.
30	SDRAM TRP parameter (TRP) - t_{RP} will be equal to 2 or 3 system clocks (0/1).
29 : 27	SDRAM TRFC parameter (SDRAM TRFC) - t_{RFC} will be equal to 3+field-value system clocks.
26	SDRAM TCAS parameter (TCAS) - Selects 2 or 3 cycle CAS delay (0/1). When changed, a LOAD-COMMAND-REGISTER command must be issued at the same time. Also sets RAS/CAS delay (t_{RCD}).
25 : 23	SDRAM bank size (SDRAM BANKSZ) - Sets the bank size for SDRAM chip selects (“000”=4 Mbyte, “001”=8 Mbyte, “010”=16 Mbyte.... “111”=512 Mbyte).
22 : 21	SDRAM column size (SDRAM COLSZ) - “00”=256, “01”=512, “10”=1024, “11”=4096 when bit 25:23=”111” 2048 otherwise.

Table 649. Memory configuration register 2.

20 : 19	SDRAM command (SDRAM CMD) - Writing a non-zero value will generate a SDRAM command. “01”=PRECHARGE, “10”=AUTO-REFRESH, “11”=LOAD-COMMAND-REGISTER. The field is reset after the command has been executed.
18	64-bit SDRAM data bus (D64) - Reads ‘1’ if the memory controller is configured for 64-bit SDRAM data bus width, ‘0’ otherwise. Read-only.
17	RESERVED
16	Mobile SDR support enabled. ‘1’ = Enabled, ‘0’ = Disabled (read-only)
15	RESERVED
14	SDRAM enable (SE) - Enables the SDRAM controller.
13	SRAM disable (SI) - Disables accesses RAM if bit 14 (SE) is set to ‘1’.
12 : 9	RAM bank size (RAM BANK SIZE) - Sets the size of each RAM bank (“0000”=8 kbyte, “0001”=16 kbyte, ..., “1111”=256 Mbyte).
8	RESERVED
7	RAM bus ready enable (RBRDY) - Enables bus ready signalling for the RAM area.
6	Read-modify-write enable (RMW) - Enables read-modify-write cycles for sub-word writes to 16- bit 32-bit areas with common write strobe (no byte write strobe).
5 : 4	RAM width (RAM WIDTH) - Sets the data width of the RAM area (“00”=8, “01”=16, “1X”=32).
3 : 2	RAM write waitstates (RAM WRITE WS) - Sets the number of wait states for RAM write cycles (“00”=0, “01”=1, “10”=2, “11”=3).
1 : 0	RAM read waitstates (RAM READ WS) - Sets the number of wait states for RAM read cycles (“00”=0, “01”=1, “10”=2, “11”=3).

58.13.3 Memory configuration register 3 (MCFG3)

MCFG3 is contains the reload value for the SDRAM refresh counter.

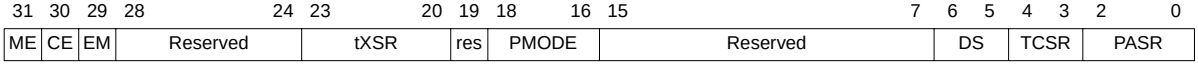


31: 27	RESERVED
26: 12	SDRAM refresh counter reload value (SDRAM REFRESH RELOAD VALUE)
11: 0	RESERVED

The period between each AUTO-REFRESH command is calculated as follows:

$t_{REFRESH}$ = ((reload value) + 1) / SYSCLK

Table 650. MCFG4 Power-Saving configuration register



31	Mobile SDRAM functionality enabled. ‘1’ = Enabled (support for Mobile SDRAM), ‘0’ = disabled (support for standard SDRAM)
30	Clock enable (CE). This value is driven on the CKE inputs of the SDRAM. Should be set to ‘1’ for correct operation. This register bit is read only when Power-Saving mode is other then none.
29	EMR. When set, the LOAD-COMMAND-REGISTER command issued by the SDRAM command filed in MCFG2 will be interpret as a LOAD-EXTENDED-COMMAND-REGISTER command.
28: 24	Reserved

Table 650. MCFG4 Power-Saving configuration register

23: 20	SDRAM tXSR timing. tXSR will be equal to field-value system clocks. (Read only when Mobile SDR support is disabled).
19	Reserved
18: 16	Power-Saving mode (Read only when Mobile SDR support is disabled). “000”: none “001”: Power-Down (PD) “010”: Self-Refresh (SR) “101”: Deep Power-Down (DPD)
15: 7	Reserved
6: 5	Selectable output drive strength (Read only when Mobile SDR support is disabled). “00”: Full “01”: One-half “10”: One-quarter “11”: Three-quarter
4: 3	Reserved for Temperature-Compensated Self Refresh (Read only when Mobile SDR support is disabled). “00”: 70ºC “01”: 45ºC “10”: 15ºC “11”: 85ºC
2: 0	Partial Array Self Refresh (Read only when Mobile SDR support is disabled). “000”: Full array (Banks 0, 1, 2 and 3) “001”: Half array (Banks 0 and 1) “010”: Quarter array (Bank 0) “101”: One-eighth array (Bank 0 with row MSB = 0) “110”: One-sixteenth array (Bank 0 with row MSB = 00)

58.14 Vendor and device identifiers

The core has vendor identifier 0x04 (ESA) and device identifier 0x00F. For description of vendor and device identifier see GRLIB IP Library User’s Manual.

58.15 Configuration options

Table 651 shows the configuration options of the core (VHDL generics).

***Table 651.*Configuration options**

Generic	Function	Allowed range	Default
hindex	AHB slave index	1 - NAHBSLV-1	0
pindex	APB slave index	0 - NAPBSLV-1	0
romaddr	ADDR field of the AHB BAR0 defining PROM address space. Default PROM area is 0x0 - 0x1FFFFFFF.	0 - 16#FFF#	16#000#
rommask	MASK field of the AHB BAR0 defining PROM address space.	0 - 16#FFF#	16#E00#
ioaddr	ADDR field of the AHB BAR1 defining I/O address space. Default I/O area is 0x20000000 - 0x2FFFFFFF.	0 - 16#FFF#	16#200#
iomask	MASK field of the AHB BAR1 defining I/O address space.	0 - 16#FFF#	16#E00#
ramaddr	ADDR field of the AHB BAR2 defining RAM address space. Default RAM area is 0x40000000-0x7FFFFFFF.	0 - 16#FFF#	16#400#
rammask	MASK field of the AHB BAR2 defining RAM address space.	0 -16#FFF#	16#C00#
paddr	ADDR field of the APB BAR configuration registers address space.	0 - 16#FFF#	0
pmask	MASK field of the APB BAR configuration registers address space.	0 - 16#FFF#	16#FFF#
wprot	RAM write protection.	0 - 1	0
invclk	Inverted clock is used for the SDRAM.	0 - 1	0
fast	Enable fast SDRAM address decoding.	0 - 1	0
romasel	log2(PROM address space size) - 1. E.g. if size of the PROM area is 0x20000000 romasel is log2(2^29)-1 = 28.	0 - 31	28
sdrasel	log2(RAM address space size) - 1. E.g if size of the RAM address space is 0x40000000 sdrasel is log2(2^30)-1= 29.	0 - 31	29
srbanks	Number of SRAM banks.	0 - 5	4
ram8	Enable 8-bit PROM and SRAM access.	0 - 1	0
ram16	Enable 16-bit PROM and SRAM access.	0 - 1	0
sden	Enable SDRAM controller.	0 - 1	0
sepbus	SDRAM is located on separate bus.	0 - 1	1
sdbits	32 or 64 -bit SDRAM data bus.	32, 64	32
oepol	Select polarity of drive signals for data pads. 0 = active low, 1 = active high.	0 - 1	0
mobile	Enable Mobile SDRAM support 0: Mobile SDRAM support disabled 1: Mobile SDRAM support enabled but not default 2: Mobile SDRAM support enabled by default 3: Mobile SDRAM support only (no regular SDR support)	0 - 3	0

58.16 Signal descriptions

Table 652 shows the interface signals of the core (VHDL ports).

***Table 652.*Signal descriptions**

Signal name	Field	Type	Function	Active
CLK	N/A	Input	Clock	-
RST	N/A	Input	Reset	Low

*Table 652.***Signal descriptions**

Signal name	Field	Type	Function	Active
MEMI	DATA[31:0]	Input	Memory data	High
	BRDYN	Input	Bus ready strobe	Low
	BEXCN	Input	Bus exception	Low
	WRN[3:0]	Input	SRAM write enable feedback signal	Low
	BWIDTH[1:0]	Input	Sets the reset value of the PROM data bus width field in the MCFG1 register	High
	SD[31:0]	Input	SDRAM separate data bus	High
MEMO	ADDRESS[31:0]	Output	Memory address	High
	DATA[31:0]	Output	Memory data	-
	SDDATA[63:0]	Output	Sdram memory data	-
	RAMSN[4:0]	Output	SRAM chip-select	Low
	RAMOEN[4:0]	Output	SRAM output enable	Low
	IOSN	Output	Local I/O select	Low
	ROMSN[1:0]	Output	PROM chip-select	Low
	OEN	Output	Output enable	Low
	WRITEN	Output	Write strobe	Low
	WRN[3:0]	Output	SRAM write enable: WRN[0] corresponds to DATA[31:24], WRN[1] corresponds to DATA[23:16], WRN[2] corresponds to DATA[15:8], WRN[3] corresponds to DATA[7:0].	Low
	MBEN[3:0]	Output	Byte enable: MBEN[0] corresponds to DATA[31:24], MBEN[1] corresponds to DATA[23:16], MBEN[2] corresponds to DATA[15:8], MBEN[3] corresponds to DATA[7:0].	Low
	BDRIVE[3:0]	Output	Drive byte lanes on external memory bus.Controls I/O-pads connected to external memory bus: BDRIVE[0] corresponds to DATA[31:24], BDRIVE[1] corresponds to DATA[23:16], BDRIVE[2] corresponds to DATA[15:8], BDRIVE[3] corresponds to DATA[7:0].	Low/High
	VBDRIVE[31:0]	Output	Vectored I/O-pad drive signals.	Low/High
	SVBDRIVE[63:0]	Output	Vectored I/O-pad drive signals for separate sdram bus.	Low/High
	READ	Output	Read strobe	High
	SA[14:0]	Output	SDRAM separate address bus	High
AHBSI	*	Input	AHB slave input signals	-
AHBSO	*	Output	AHB slave output signals	-
APBI	*	Input	APB slave input signals	-
APBO	*	Output	APB slave output signals	-
WPROT	WPROTHIT	Input	Unused	-

*Table 652.***Signal descriptions**

Signal name	Field	Type	Function	Active
SDO	SDCASN	Output	SDRAM column address strobe	Low
	SDCKE[1:0]	Output	SDRAM clock enable	High
	SDCSN[1:0]	Output	SDRAM chip select	Low
	SDDQM[7:0]	Output	SDRAM data mask: DQM[7] corresponds to DATA[63:56], DQM[6] corresponds to DATA[55:48], DQM[5] corresponds to DATA[47:40], DQM[4] corresponds to DATA[39:32], DQM[3] corresponds to DATA[31:24], DQM[2] corresponds to DATA[23:16], DQM[1] corresponds to DATA[15:8], DQM[0] corresponds to DATA[7:0].	Low
	SDRASN	Output	SDRAM row address strobe	Low
	SDWEN	Output	SDRAM write enable	Low

* see GRLIB IP Library User's Manual

58.17 Library dependencies

Table 653 shows libraries used when instantiating the core (VHDL libraries).

*Table 653.***Library dependencies**

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AHB signal definitions
GAISLER	MEMCTRL	Signals	Memory bus signals definitions
		Components	SDMCTRL component
ESA	MEMORYCTRL	Component	Memory controller component declaration

58.18 Instantiation

This example shows how the core can be instantiated.

The example design contains an AMBA bus with a number of AHB components connected to it including the memory controller. The external memory bus is defined on the example designs port map and connected to the memory controller. System clock and reset are generated by GR Clock Generator and Reset Generator.

Memory controller decodes default memory areas: PROM area is 0x0 - 0x1FFFFFFF, I/O-area is 0x20000000-0x3FFFFFFF and RAM area is 0x40000000 - 0x7FFFFFFF. SDRAM controller is enabled. SDRAM clock is synchronized with system clock by clock generator.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
use grlib.tech.all;
library gaisler;
use gaisler.memctrl.all;
use gaisler.pads.all;   -- used for I/O pads
library esa;
use esa.memoryctrl.all;

entity mctrl_ex is
```

```
  port (
    clk : in std_ulogic;
    resetn : in std_ulogic;
    pllref : in  std_ulogic;

    -- memory bus
    address  : out   std_logic_vector(27 downto 0); -- memory bus
    data     : inout std_logic_vector(31 downto 0);
    ramsn    : out   std_logic_vector(4 downto 0);
    ramoen   : out   std_logic_vector(4 downto 0);
    rwen     : inout std_logic_vector(3 downto 0);
    romsn    : out   std_logic_vector(1 downto 0);
    iosn     : out   std_logic;
    oen      : out   std_logic;
    read     : out   std_logic;
    writen   : inout std_logic;
    brdyn    : in    std_logic;
    bexcn    : in    std_logic;
-- sdram i/f
    sdcke    : out std_logic_vector ( 1 downto 0);  -- clk en
    sdcsn    : out std_logic_vector ( 1 downto 0);  -- chip sel
    sdwen    : out std_logic;                       -- write en
    sdrasn   : out std_logic;                       -- row addr stb
    sdcasn   : out std_logic;                       -- col addr stb
    sddqm    : out std_logic_vector (7 downto 0);  -- data i/o mask
    sdclk    : out std_logic;                       -- sdram clk output
    sa       : out std_logic_vector(14 downto 0); -- optional sdram address
    sd       : inout std_logic_vector(63 downto 0) -- optional sdram data
    );
end;

architecture rtl of mctrl_ex is

  -- AMBA bus (AHB and APB)
  signal apbi  : apb_slv_in_type;
  signal apbo  : apb_slv_out_vector := (others => apb_none);
  signal ahbsi : ahb_slv_in_type;
  signal ahbso : ahb_slv_out_vector := (others => ahbs_none);
  signal ahbmi : ahb_mst_in_type;
  signal ahbmo : ahb_mst_out_vector := (others => ahbm_none);

  -- signals used to connect memory controller and memory bus
  signal memi : memory_in_type;
  signal memo : memory_out_type;

  signal sdo : sdram_out_type;

  signal wprot : wprot_out_type;  -- dummy signal, not used
  signal clkm, rstn : std_ulogic; -- system clock and reset

-- signals used by clock and reset generators
  signal cgi : clkgen_in_type;
  signal cgo : clkgen_out_type;

  signal gnd : std_ulogic;

begin

  -- Clock and reset generators
  clkgen0 : clkgen generic map (clk_mul => 2, clk_div => 2, sdramen => 1,
                                tech => virtex2, sdinvclk => 0)
  port map (clk, gnd, clkm, open, open, sdclk, open, cgi, cgo);

  cgi.pllctrl <= "00"; cgi.pllrst <= resetn; cgi.pllref <= pllref;

  -- Memory controller
  mctrl0 : mctrl generic map (srbanks => 1, sden => 1)
    port map (rstn, clkm, memi, memo, ahbsi, ahbso(0), apbi, apbo(0), wprot, sdo);

  -- memory controller inputs not used in this configuration
  memi.brdyn <= '1'; memi.bexcn <= '1'; memi.wrn <= "1111";
```

```
  memi.sd <= sd;

  -- prom width at reset
  memi.bwidth <= "10";

  -- I/O pads driving data memory bus data signals
  datapads : for i in 0 to 3 generate
      data_pad : iopadv generic map (width => 8)
      port map (pad => data(31-i*8 downto 24-i*8),
                o => memi.data(31-i*8 downto 24-i*8),
                en => memo.bdrive(i),
                i => memo.data(31-i*8 downto 24-i*8));
  end generate;

  -- connect memory controller outputs to entity output signals
  address <= memo.address; ramsn <= memo.ramsn; romsn <= memo.romsn;
  oen <= memo.oen; rwen <= memo.wrn; ramoen <= "1111" & memo.ramoen(0);
  sa <= memo.sa;
  writen <= memo.writen; read <= memo.read; iosn <= memo.iosn;
  sdcke <= sdo.sdcke; sdwen <= sdo.sdwen; sdcsn <= sdo.sdcsn;
  sdrasn <= sdo.rasn; sdcasn <= sdo.casn; sddqm <= sdo.dqm;
end;
```

Beilage 8

GRETH GRLIB Auszug

40 GRETH - Ethernet Media Access Controller (MAC) with EDCL support

40.1 Overview

Gaisler Research's Ethernet Media Access Controller (GRETH) provides an interface between an AMBA-AHB bus and an Ethernet network. It supports 10/100 Mbit speed in both full- and half-duplex. The AMBA interface consists of an APB interface for configuration and control and an AHB master interface which handles the dataflow. The dataflow is handled through DMA channels. There is one DMA engine for the transmitter and one for the receiver. Both share the same AHB master interface. The ethernet interface supports both the MII and RMII interfaces which should be connected to an external PHY. The GRETH also provides access to the MII Management interface which is used to configure the PHY.

Optional hardware support for the Ethernet Debug Communication Link (EDCL) protocol is also provided. This is an UDP/IP based protocol used for remote debugging.

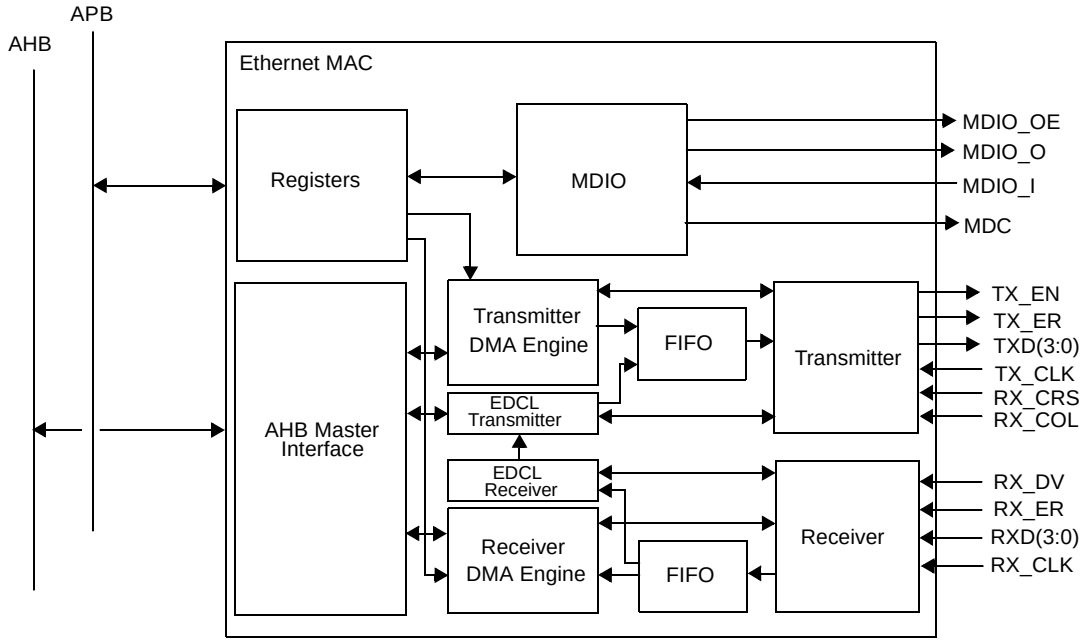


Figure 136. Block diagram of the internal structure of the GRETH.

40.2 Operation

40.2.1 System overview

The GRETH consists of 3 functional units: The DMA channels, MDIO interface and the optional Ethernet Debug Communication Link (EDCL).

The main functionality consists of the DMA channels which are used to transfer data between an AHB bus and an Ethernet network. There is one transmitter DMA channel and one Receiver DMA channel. The operation of the DMA channels is controlled through registers accessible through the APB interface.

The MDIO interface is used for accessing configuration and status registers in one or more PHYs connected to the MAC. The operation of this interface is also controlled through the APB interface.

The optional EDCL provides read and write access to an AHB bus through Ethernet. It uses the UDP, IP, ARP protocols together with a custom application layer protocol to accomplish this. The EDCL contains no user accessible registers and always runs in parallel with the DMA channels.

The Media Independent Interface (MII) is used for communicating with the PHY. There is an Ethernet transmitter which sends all data from the AHB domain on the Ethernet using the MII interface. Correspondingly, there is an Ethernet receiver which stores all data from the Ethernet on the AHB bus. Both of these interfaces use FIFOs when transferring the data streams. The GRETH also supports the RMII which uses a subset of the MII signals.

The EDCL and the DMA channels share the Ethernet receiver and transmitter.

40.2.2 Protocol support

The GRETH is implemented according to IEEE standard 802.3-2002. There is no support for the optional control sublayer and no multicast addresses can be assigned to the MAC. This means that packets with type 0x8808 (the only currently defined ctrl packets) are discarded.

40.2.3 Clocking

GRETH has three clock domains: The AHB clock, Ethernet receiver clock and the Ethernet transmitter clock. The ethernet transmitter and receiver clocks are generated by the external ethernet PHY, and are inputs to the core through the MII interface. The three clock domains are unrelated to each other and all signals crossing the clock regions are fully synchronized inside the core.

Both full-duplex and half-duplex operating modes are supported and both can be run in either 10 or 100 Mbit. The minimum AHB clock for 10 Mbit operation is 2.5 MHz, while 18 MHz is needed for 100 Mbit. Using a lower AHB clock than specified will lead to excessive packet loss.

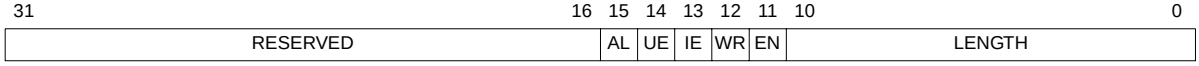
40.3 Tx DMA interface

The transmitter DMA interface is used for transmitting data on an Ethernet network. The transmission is done using descriptors located in memory.

40.3.1 Setting up a descriptor.

A single descriptor is shown in table 348 and 349. The number of bytes to be sent should be set in the length field and the address field should point to the data. The address must be word-aligned. If the interrupt enable (IE) bit is set, an interrupt will be generated when the packet has been sent (this requires that the transmitter interrupt bit in the control register is also set). The interrupt will be generated regardless of whether the packet was transmitted successfully or not. The Wrap (WR) bit is also a control bit that should be set before transmission and it will be explained later in this section.

Table 348. GRETH transmit descriptor word 0 (address offset 0x0)



31: 16	RESERVED
15	Attempt Limit Error (AL) - The packet was not transmitted because the maximum number of attempts was reached.
14	Underrun Error (UE) - The packet was incorrectly transmitted due to a FIFO underrun error.

Table 348. GRETH transmit descriptor word 0 (address offset 0x0)

13	Interrupt Enable (IE) - Enable Interrupts. An interrupt will be generated when the packet from this descriptor has been sent provided that the transmitter interrupt enable bit in the control register is set. The interrupt is generated regardless if the packet was transmitted successfully or if it terminated with an error.
12	Wrap (WR) - Set to one to make the descriptor pointer wrap to zero after this descriptor has been used. If this bit is not set the pointer will increment by 8. The pointer automatically wraps to zero when the 1 kB boundary of the descriptor table is reached.
11	Enable (EN) - Set to one to enable the descriptor. Should always be set last of all the descriptor fields.
10: 0	LENGTH - The number of bytes to be transmitted.

Table 349. GRETH transmit descriptor word 1 (address offset 0x4)

31	2	1	0
ADDRESS		RES	

31: 2	Address (ADDRESS) - Pointer to the buffer area from where the packet data will be loaded.
1: 0	RESERVED

To enable a descriptor the enable (EN) bit should be set and after this is done, the descriptor should not be touched until the enable bit has been cleared by the GRETH.

40.3.2 Starting transmissions

Enabling a descriptor is not enough to start a transmission. A pointer to the memory area holding the descriptors must first be set in the GRETH. This is done in the transmitter descriptor pointer register. The address must be aligned to a 1 kB boundary. Bits 31 to 10 hold the base address of descriptor area while bits 9 to 3 form a pointer to an individual descriptor.The first descriptor should be located at the base address and when it has been used by the GRETH the pointer field is incremented by 8 to point at the next descriptor. The pointer will automatically wrap back to zero when the next 1 kB boundary has been reached (the descriptor at address offset 0x3F8 has been used). The WR bit in the descriptors can be set to make the pointer wrap back to zero before the 1 kB boundary.

The pointer field has also been made writable for maximum flexibility but care should be taken when writing to the descriptor pointer register. It should never be touched when a transmission is active.

The final step to activate the transmission is to set the transmit enable bit in the control register. This tells the GRETH that there are more active descriptors in the descriptor table. This bit should always be set when new descriptors are enabled, even if transmissions are already active. The descriptors must always be enabled before the transmit enable bit is set.

40.3.3 Descriptor handling after transmission

When a transmission of a packet has finished, status is written to the first word in the corresponding descriptor. The Underrun Error bit is set if the FIFO became empty before the packet was completely transmitted while the Attempt Limit Error bit is set if more collisions occurred than allowed. The packet was successfully transmitted only if both of these bits are zero. The other bits in the first descriptor word are set to zero after transmission while the second word is left untouched.

The enable bit should be used as the indicator when a descriptor can be used again, which is when it has been cleared by the GRETH. There are three bits in the GRETH status register that hold transmission status. The Transmitter Error (TE) bit is set each time an transmission ended with an error (when at least one of the two status bits in the transmit descriptor has been set). The Transmitter Interrupt (TI) is set each time a transmission ended successfully.

The transmitter AHB error (TA) bit is set when an AHB error was encountered either when reading a descriptor or when reading packet data. Any active transmissions were aborted and the transmitter was disabled. The transmitter can be activated again by setting the transmit enable register.

40.3.4 Setting up the data for transmission

The data to be transmitted should be placed beginning at the address pointed by the descriptor address field. The GRETH does not add the Ethernet address and type fields so they must also be stored in the data buffer. The 4 B Ethernet CRC is automatically appended at the end of each packet. Each descriptor will be sent as a single Ethernet packet. If the size field in a descriptor is greater than 1514 B, the packet will not be sent.

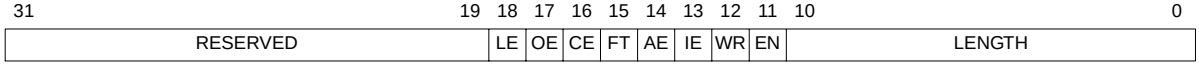
40.4 Rx DMA interface

The receiver DMA interface is used for receiving data from an Ethernet network. The reception is done using descriptors located in memory.

40.4.1 Setting up descriptors

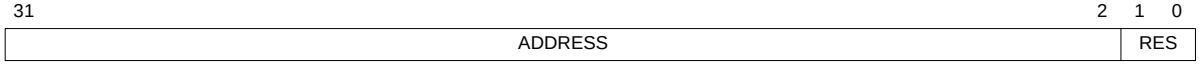
A single descriptor is shown in table 350 and 351. The address field should point to a word-aligned buffer where the received data should be stored. The GRETH will never store more than 1514 B to the buffer. If the interrupt enable (IE) bit is set, an interrupt will be generated when a packet has been received to this buffer (this requires that the receiver interrupt bit in the control register is also set). The interrupt will be generated regardless of whether the packet was received successfully or not. The Wrap (WR) bit is also a control bit that should be set before the descriptor is enabled and it will be explained later in this section.

Table 350. GRETH receive descriptor word 0 (address offset 0x0)



31: 19	RESERVED
18	Length error (LE) - The length/type field of the packet did not match the actual number of received bytes.
17	Overrun error (OE) - The frame was incorrectly received due to a FIFO overrun.
16	CRC error (CE) - A CRC error was detected in this frame.
15	Frame too long (FT) - A frame larger than the maximum size was received. The excessive part was truncated.
14	Alignment error (AE) - An odd number of nibbles were received.
13	Interrupt Enable (IE) - Enable Interrupts. An interrupt will be generated when a packet has been received to this descriptor provided that the receiver interrupt enable bit in the control register is set. The interrupt is generated regardless if the packet was received successfully or if it terminated with an error.
12	Wrap (WR) - Set to one to make the descriptor pointer wrap to zero after this descriptor has been used. If this bit is not set the pointer will increment by 8. The pointer automatically wraps to zero when the 1 kB boundary of the descriptor table is reached.
11	Enable (EN) - Set to one to enable the descriptor. Should always be set last of all the descriptor fields.
10: 0	LENGTH - The number of bytes received to this descriptor.

Table 351. GRETH receive descriptor word 1 (address offset 0x4)



31: 2	Address (ADDRESS) - Pointer to the buffer area from where the packet data will be loaded.
1: 0	RESERVED

40.4.2 Starting reception

Enabling a descriptor is not enough to start reception. A pointer to the memory area holding the descriptors must first be set in the GRETH. This is done in the receiver descriptor pointer register. The address must be aligned to a 1 kB boundary. Bits 31 to 10 hold the base address of descriptor area while bits 9 to 3 form a pointer to an individual descriptor. The first descriptor should be located at the base address and when it has been used by the GRETH the pointer field is incremented by 8 to point at the next descriptor. The pointer will automatically wrap back to zero when the next 1 kB boundary has been reached (the descriptor at address offset 0x3F8 has been used). The WR bit in the descriptors can be set to make the pointer wrap back to zero before the 1 kB boundary.

The pointer field has also been made writable for maximum flexibility but care should be taken when writing to the descriptor pointer register. It should never be touched when reception is active.

The final step to activate reception is to set the receiver enable bit in the control register. This will make the GRETH read the first descriptor and wait for an incoming packet.

40.4.3 Descriptor handling after reception

The GRETH indicates a completed reception by clearing the descriptor enable bit. The other control bits (WR, IE) are also cleared. The number of received bytes is shown in the length field. The parts of the Ethernet frame stored are the destination address, source address, type and data fields. Bits 17-14 in the first descriptor word are status bits indicating different receive errors. All four bits are zero after a reception without errors. The status bits are described in table 350.

Packets arriving that are smaller than the minimum Ethernet size of 64 B are not considered as a reception and are discarded. The current receive descriptor will be left untouched an used for the first packet arriving with an accepted size. The TS bit in the status register is set each time this event occurs.

If a packet is received with an address not accepted by the MAC, the IA status register bit will be set.

Packets larger than maximum size cause the FT bit in the receive descriptor to be set. The length field is not guaranteed to hold the correct value of received bytes. The counting stops after the word containing the last byte up to the maximum size limit has been written to memory.

The address word of the descriptor is never touched by the GRETH.

40.4.4 Reception with AHB errors

If an AHB error occurs during a descriptor read or data store, the Receiver AHB Error (RA) bit in the status register will be set and the receiver is disabled. The current reception is aborted. The receiver can be enabled again by setting the Receive Enable bit in the control register.

40.5 MDIO Interface

The MDIO interface provides access to PHY configuration and status registers through a two-wire interface which is included in the MII interface. The GRETH provided full support for the MDIO interface. If it is not needed in a design it can be removed with a VHDL generic.

The MDIO interface can be used to access from 1 to 32 PHY containing 1 to 32 16-bit registers. A read transfer i set up by writing the PHY and register addresses to the MDIO Control register and setting the read bit. This caused the Busy bit to be set and the operation is finished when the Busy bit is cleared. If the operation was successful the Linkfail bit is zero and the data field contains the read data. An unsuccessful operation is indicated by the Linkfail bit being set. The data field is undefined in this case.

A write operation is started by writing the 16-bit data, PHY address and register address to the MDIO Control register and setting the write bit. The operation is finished when the busy bit is cleared and it was successful if the Linkfail bit is zero.

40.5.1 PHY interrupts

The core also supports status change interrupts from the PHY. A level sensitive interrupt signal can be connected on the mdint input. The mdint_pol vhdl generic can be used to select the polarity. The PHY status change bit in the status register is set each time an event is detected in this signal. If the PHY status interrupt enable bit is set at the time of the event the core will also generate an interrupt on the AHB bus.

40.6 Ethernet Debug Communication Link (EDCL)

The EDCL provides access to an on-chip AHB bus through Ethernet. It uses the UDP, IP and ARP protocols together with a custom application layer protocol. The application layer protocol uses an ARQ algorithm to provide reliable AHB instruction transfers. Through this link, a read or write transfer can be generated to any address on the AHB bus. The EDCL is optional and must be enabled with a generic.

40.6.1 Operation

The EDCL receives packets in parallel with the MAC receive DMA channel. It uses a separate MAC address which is used for distinguishing EDCL packets from packets destined to the MAC DMA channel. The EDCL also has an IP address which is set through generics. Since ARP packets use the Ethernet broadcast address, the IP-address must be used in this case to distinguish between EDCL ARP packets and those that should go to the DMA-channel. Packets that are determined to be EDCL packets are not processed by the receive DMA channel.

When the packets are checked to be correct, the AHB operation is performed. The operation is performed with the same AHB master interface that the DMA-engines use. The replies are automatically sent by the EDCL transmitter when the operation is finished. It shares the Ethernet transmitter with the transmitter DMA-engine but has higher priority.

40.6.2 EDCL protocols

The EDCL accepts Ethernet frames containing IP or ARP data. ARP is handled according to the protocol specification with no exceptions.

IP packets carry the actual AHB commands. The EDCL expects an Ethernet frame containing IP, UDP and the EDCL specific application layer parts. Table 352 shows the IP packet required by the EDCL. The contents of the different protocol headers can be found in TCP/IP literature.

*Table 352.*The IP packet expected by the EDCL.

Ethernet Header	IP Header	UDP Header	2 B Offset	4 B Control word	4 B Address	Data 0 - 242 4B Words	Ethernet CRC

The following is required for successful communication with the EDCL: A correct destination MAC address as set by the generics, an Ethernet type field containing 0x0806 (ARP) or 0x0800 (IP). The IP-address is then compared with the value determined by the generics for a match. The IP-header checksum and identification fields are not checked. There are a few restrictions on the IP-header fields. The version must be four and the header size must be 5 B (no options). The protocol field must always be 0x11 indicating a UDP packet. The length and checksum are the only IP fields changed for the reply.

The EDCL only provides one service at the moment and it is therefore not required to check the UDP port number. The reply will have the original source port number in both the source and destination fields. UDP checksum are not used and the checksum field is set to zero in the replies.

The UDP data field contains the EDCL application protocol fields. Table 353 shows the application protocol fields (data field excluded) in packets received by the EDCL. The 16-bit offset is used to align the rest of the application layer data to word boundaries in memory and can thus be set to any

value. The R/W field determines whether a read (0) or a write(1) should be performed. The length

*Table 353.*The EDCL application layer fields in received frames.

16-bit Offset	14-bit Sequence number	1-bit R/W	10-bit Length	7-bit Unused

field contains the number of bytes to be read or written. If R/W is one the data field shown in table 352 contains the data to be written. If R/W is zero the data field is empty in the received packets. Table 354 shows the application layer fields of the replies from the EDCL. The length field is always zero for replies to write requests. For read requests it contains the number of bytes of data contained in the data field.

*Table 354.*The EDCL application layer fields in transmitted frames.

16-bit Offset	14-bit sequence number	1-bit ACK/NAK	10-bit Length	7-bit Unused

The EDCL implements a Go-Back-N algorithm providing reliable transfers. The 14-bit sequence number in received packets are checked against an internal counter for a match. If they do not match, no operation is performed and the ACK/NAK field is set to 1 in the reply frame. The reply frame contains the internal counter value in the sequence number field. If the sequence number matches, the operation is performed, the internal counter is incremented, the internal counter value is stored in the sequence number field and the ACK/NAK field is set to 0 in the reply. The length field is always set to 0 for ACK/NAK=1 frames. The unused field is not checked and is copied to the reply. It can thus be set to hold for example some extra identifier bits if needed.

40.6.3 EDCL IP and Ethernet address settings

The default value of the EDCL IP and MAC addresses are set by `ipaddrh, ipaddrl, macaddrh and macaddrl` generics. The IP address can later be changed by software, but the MAC address is fixed. To allow several EDCL enabled GRETH controllers on the same sub-net, the 4 LSB bits of the IP and MAC address can optionally be set by an input signal. This is enabled by setting the `edcl` generic = 2, and driving the 4-bit LSB value on ethi.edcladdr.

40.6.4 EDCL limitations

The EDCL is designed to work without software intervention thus requiring automatic configuration in hardware. To simplify the hardware the EDCL is assumed to be able to handle all operating modes of the PHY and only reads the auto-negotiated mode and configures the MAC. This prevents it from working with gigabit capable PHYs since they might be in the gigabit mode and this cannot be handled by the MAC.

40.7 Media Independent Interfaces

There are several interfaces defined between the MAC sublayer and the Physical layer. The GRETH supports two of them: The Media Independent Interface (MII) and the Reduced Media Independent Interface (RMII).

The MII was defined in the 802.3 standard and is most commonly supported. The ethernet interface have been implemented according to this specification. It uses 16 signals.

The RMII was developed to meet the need for an interface allowing Ethernet controllers with smaller pin counts. It uses 6 (7) signals which are a subset of the MII signals. Table 355 shows the mapping between the RMII signals and the GRLIB MII interface.

*Table 355.*Signal mappings between RMII and the GRLIB MII interface.

RMII	MII
txd[1:0]	txd[1:0]
tx_en	tx_en
crs_dv	rx_crs
rxd[1:0]	rxd[1:0]
ref_clk	rmii_clk
rx_er	not used

40.8 Software drivers

Drivers for the GRETH MAC is provided for the following operating systems: RTEMS, eCos, uClinux and Linux-2.6. The drivers are freely available in full source code under the GPL license from Gaisler Research's web site (http://gaisler.com/).

40.9 Registers

The core is programmed through registers mapped into APB address space.

*Table 356.*GRETH registers

APB address offset	**Register**
0x0	Control register
0x4	Status/Interrupt-source register
0x8	MAC Address MSB
0xC	MAC Address LSB
0x10	MDIO Control/Status
0x14	Transmit descriptor pointer
0x18	Receiver descriptor pointer
0x1C	EDCL IP

Table 357. GRETH control register

31	30 28	27 11	10	9 8	7	6	5	4	3	2	1	0
ED	BS	RESERVED	PI	RES	SP	RS	PM	FD	RI	TI	RE	TE

31	EDCL available (ED) - Set to one if the EDCL is available.
30: 28	EDCL buffer size (BS) - Shows the amount of memory used for EDCL buffers. 0 = 1 kB, 1 = 2 kB, ..., 6 = 64 kB.
27: 11	RESERVED
10	PHY status change interrupt enable (PI) - Enables interrupts for detected PHY status changes.
9: 8	RESERVED
7	Speed (SP) - Sets the current speed mode. 0 = 10 Mbit, 1 = 100 Mbit. Only used in RMII mode (rmii = 1). A default value is automatically read from the PHY after reset. Reset value: '1'.
6	Reset (RS) - A one written to this bit resets the GRETH core. Self clearing.
5	Promiscuous mode (PM) - If set, the GRETH operates in promiscuous mode which means it will receive all packets regardless of the destination address. Reset value: '0'.

Table 357. GRETH control register

4	Full duplex (FD) - If set, the GRETH operates in full-duplex mode otherwise it operates in half-duplex. Reset value: ‘0’.
3	Receiver interrupt (RI) - Enable Receiver Interrupts. An interrupt will be generated each time a packet is received when this bit is set. The interrupt is generated regardless if the packet was received successfully or if it terminated with an error. Reset value: ‘0’.
2	Transmitter interrupt (TI) - Enable Transmitter Interrupts. An interrupt will be generated each time a packet is transmitted when this bit is set. The interrupt is generated regardless if the packet was transmitted successfully or if it terminated with an error. Reset value: ‘0’.
1	Receive enable (RE) - Should be written with a one each time new descriptors are enabled. As long as this bit is one the GRETH will read new descriptors and as soon as it encounters a disabled descriptor it will stop until RE is set again. This bit should be written with a one after the new descriptors have been enabled. Reset value: ‘0’.
0	Transmit enable (TE) - Should be written with a one each time new descriptors are enabled. As long as this bit is one the GRETH will read new descriptors and as soon as it encounters a disabled descriptor it will stop until TE is set again. This bit should be written with a one after the new descriptors have been enabled. Reset value: ‘0’.

Table 358. GRETH status register

31 … 9	8	7	6	5	4	3	2	1	0
RESERVED	PS	IA	TS	TA	RA	TI	RI	TE	RE

8	PHY status changes (PS) - Set each time a PHY status change is detected.
7	Invalid address (IA) - A packet with an address not accepted by the MAC was received. Cleared when written with a one. Reset value: ‘0’.
6	Too small (TS) - A packet smaller than the minimum size was received. Cleared when written with a one. Reset value: ‘0’.
5	Transmitter AHB error (TA) - An AHB error was encountered in transmitter DMA engine. Cleared when written with a one. Not Reset.
4	Receiver AHB error (RA) - An AHB error was encountered in receiver DMA engine. Cleared when written with a one. Not Reset.
3	Transmitter interrupt (TI) - A packet was transmitted without errors. Cleared when written with a one. Not Reset.
2	Receiver interrupt (RI) - A packet was received without errors. Cleared when written with a one. Not Reset.
1	Transmitter error (TE) - A packet was transmitted which terminated with an error. Cleared when written with a one. Not Reset.
0	Receiver error (RE) - A packet has been received which terminated with an error. Cleared when written with a one. Not Reset.

Table 359. GRETH MAC address MSB.

31 … 16	15 … 0
RESERVED	Bit 47 downto 32 of the MAC address

31: 16	RESERVED
15: 0	The two most significant bytes of the MAC Address. Not Reset.

Table 360. GRETH MAC address LSB.

31	0
Bit 31 downto 0 of the MAC address	

31: 0 The four least significant bytes of the MAC Address. Not Reset.

Table 361. GRETH MDIO ctrl/status register.

31 16	15 11	10 6	5	4	3	2	1	0
DATA	PHYADDR	REGADDR		NV	BU	LF	RD	WR

31: 16 Data (DATA) - Contains data read during a read operation and data that is transmitted is taken from this field. Reset value: 0x0000.

15: 11 PHY address (PHYADDR) - This field contains the address of the PHY that should be accessed during a write or read operation. Reset value: “00000”.

10: 6 Register address (REGADDR) - This field contains the address of the register that should be accessed during a write or read operation. Reset value: “00000”.

5 RESERVED

4 Not valid (NV) - When an operation is finished (BUSY = 0) this bit indicates whether valid data has been received that is, the data field contains correct data. Reset value: ‘0’.

3 Busy (BU) - When an operation is performed this bit is set to one. As soon as the operation is finished and the management link is idle this bit is cleared. Reset value: ‘0’.

2 Linkfail (LF) - When an operation completes (BUSY = 0) this bit is set if a functional management link was not detected. Reset value: ‘1’.

1 Read (RD) - Start a read operation on the management interface. Data is stored in the data field. Reset value: ‘0’.

0 Write (WR) - Start a write operation on the management interface. Data is taken from the Data field. Reset value: ‘0’.

Table 362. GRETH transmitter descriptor table base address register.

31 10	9 3	2 0
BASEADDR	DESCPNT	RES

31: 10 Transmitter descriptor table base address (BASEADDR) - Base address to the transmitter descriptor table.Not Reset.

9: 3 Descriptor pointer (DESCPNT) - Pointer to individual descriptors. Automatically incremented by the Ethernet MAC.

2: 0 RESERVED

Table 363. GRETH receiver descriptor table base address register.

31 10	9 3	2 0
BASEADDR	DESCPNT	RES

31: 10 Receiver descriptor table base address (BASEADDR) - Base address to the receiver descriptor table.Not Reset.

9: 3 Descriptor pointer (DESCPNT) - Pointer to individual descriptors. Automatically incremented by the Ethernet MAC.

2: 0 RESERVED

Table 364. GRETH EDCL IP register

31: 0 EDCL IP address. Reset value is set with the ipaddrh and ipaddrl generics.

40.10 Vendor and device identifiers

The core has vendor identifier 0x01 (Gaisler Research) and device identifier 0x1D. For description of vendor and device identifiers see GRLIB IP Library User’s Manual.

40.11 Configuration options

Table 365 shows the configuration options of the core (VHDL generics).

*Table 365.*Configuration options

Generic	Function	Allowed range	Default
hindex	AHB master index.	0 - NAHBMST-1	0
pindex	APB slave index	0 - NAPBSLV-1	0
paddr	Addr field of the APB bar.	0 - 16#FFF#	0
pmask	Mask field of the APB bar.	0 - 16#FFF#	16#FFF#
pirq	Interrupt line used by the GRETH.	0 - NAHBIRQ-1	0
memtech	Memory technology used for the FIFOs.	0 - NTECH	inferred
ifg_gap	Number of ethernet clock cycles used for one interframe gap. Default value as required by the standard. Do not change unless you know what your doing.	1 - 255	24
attempt_limit	Maximum number of transmission attempts for one packet. Default value as required by the standard.	1 - 255	16
backoff_limit	Limit on the backoff size of the backoff time. Default value as required by the standard. Sets the number of bits used for the random value. Do not change unless you know what your doing.	1 - 10	10
slot_time	Number of ethernet clock cycles used for one slot- time. Default value as required by the ethernet standard. Do not change unless you know what you are doing.	1 - 255	128
mdcscaler	Sets the divisor value use to generate the mdio clock (mdc). The mdc frequency will be clk/(2*(mdcs-caler+1)).	0 - 255	25
enable_mdio	Enable the Management interface,	0 - 1	0
fifosize	Sets the size in 32-bit words of the receiver and transmitter FIFOs.	4 - 32	8
nsync	Number of synchronization registers used.	1 - 2	2
edcl	Enable EDCL. 0 = disabled. 1 = enabled. 2 = enabled and 4-bit LSB of IP and ethernet MAC address programmed by ethi.edcladdr.	0 - 2	0
edclbufsz	Select the size of the EDCL buffer in kB.	1 - 64	1
macaddrh	Sets the upper 24 bits of the EDCL MAC address.*)	0 - 16#FFFFFF#	16#00005E#
macaddrl	Sets the lower 24 bits of the EDCL MAC address. *)	0 - 16#FFFFFF#	16#000000#
ipaddrh	Sets the upper 16 bits of the EDCL IP address reset value.	0 - 16#FFFF#	16#C0A8#
ipaddrl	Sets the lower 16 bits of the EDCL IP address reset value.	0 - 16#FFFF#	16#0035#
phyrstadr	Sets the reset value of the PHY address field in the MDIO register.	0 - 31	0
rmii	Selects the desired PHY interface. 0 = MII, 1 = RMII.	0 - 1	0
oepol	Selects polarity on output enable (ETHO.MDIO_OE). 0 = active low, 1 = active high	0 - 1	0
mdint_pol	Selects polarity for level sensitive PHY interrupt line. 0 = active low, 1 = active high	0 - 1	0

*) Not all addresses are allowed and most NICs and protocol implementations will discard frames with illegal addresses silently. Consult network literature if unsure about the addresses.

40.12 Signal descriptions

Table 366 shows the interface signals of the core (VHDL ports).

*Table 366.*Signal descriptions

Signal name	Field	Type	Function	Active
RST	N/A	Input	Reset	Low
CLK	N/A	Input	Clock	-
AHBMI	*	Input	AMB master input signals	-
AHBMO	*	Output	AHB master output signals	-
APBI	*	Input	APB slave input signals	-
APBO	*	Output	APB slave output signals	-
ETHI	gtx_clk	Input	Ethernet gigabit transmit clock.	-
	rmii_clk	Input	Ethernet RMII clock.	-
	tx_clk	Input	Ethernet transmit clock.	-
	rx_clk	Input	Ethernet receive clock.	-
	rxd	Input	Ethernet receive data.	-
	rx_dv	Input	Ethernet receive data valid.	High
	rx_er	Input	Ethernet receive error.	High
	rx_col	Input	Ethernet collision detected. (Asynchronous, sampled with tx_clk)	High
	rx_crs	Input	Ethernet carrier sense. (Asynchronous, sampled with tx_clk)	High
	mdio_i	Input	Ethernet management data input	-
	phyrstaddr	Input	Reset address for GRETH PHY address field.	-
	edcladdr	Input	Sets the four least significant bits of the EDCL MAC address when the edcl generic is set to 2.	-
ETHO	reset	Output	Ethernet reset (asserted when the MAC is reset).	Low
	txd	Output	Ethernet transmit data.	-
	tx_en	Output	Ethernet transmit enable.	High
	tx_er	Output	Ethernet transmit error.	High
	mdc	Output	Ethernet management data clock.	-
	mdio_o	Output	Ethernet management data output.	-
	mdio_oe	Output	Ethernet management data output enable.	Set by the oepol generic.

* see GRLIB IP Library User's Manual

40.13 Library dependencies

Table 367 shows libraries used when instantiating the core (VHDL libraries).

*Table 367.*Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AMBA signal definitions
GAISLER	NET	Signals, components	GRETH component declaration

40.14 Instantiation

This example shows how the core can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
use grlib.tech.all;
library gaisler;
use gaisler.ethernet_mac.all;

entity greth_ex is
  port (
    clk  : in std_ulogic;
    rstn : in std_ulogic;

    -- ethernet signals
    ethi :: in  eth_in_type;
    etho :  in  eth_out_type
    );
end;

architecture rtl of greth_ex is

  -- AMBA signals
  signal apbi  : apb_slv_in_type;
  signal apbo  : apb_slv_out_vector := (others => apb_none);
  signal ahbmi : ahb_mst_in_type;
  signal ahbmo : ahb_mst_out_vector := (others => ahbm_none);

begin

  -- AMBA Components are instantiated here
  ...

  -- GRETH
  e1 : greth
    generic map(
     hindex       => 0,
     pindex       => 12,
     paddr        => 12,
     pirq         => 12,
     memtech      => inferred,
     mdcscaler    => 50,
     enable_mdio  => 1,
     fifosize     => 32,
     nsync        => 1,
     edcl         => 1,
     edclbufsz    => 8,
     macaddrh     => 16#00005E#,
     macaddrl     => 16#00005D#,
     ipaddrh      => 16#c0a8#,
     ipaddrl      => 16#0035#)
 port map(
   rst          => rstn,
   clk          => clk,
   ahbmi        => ahbmi,
   ahbmo        => ahbmo(0),
   apbi         => apbi,
   apbo         => apbo(12),
   ethi         => ethi,
   etho         => etho
   );
end;
```

Beilage 9

AHBUART GRLIB Auszug

12 AHBUART- AMBA AHB Serial Debug Interface

12.1 Overview

The interface consists of a UART connected to the AMBA AHB bus as a master. A simple communication protocol is supported to transmit access parameters and data. Through the communication link, a read or write transfer can be generated to any address on the AMBA AHB bus.

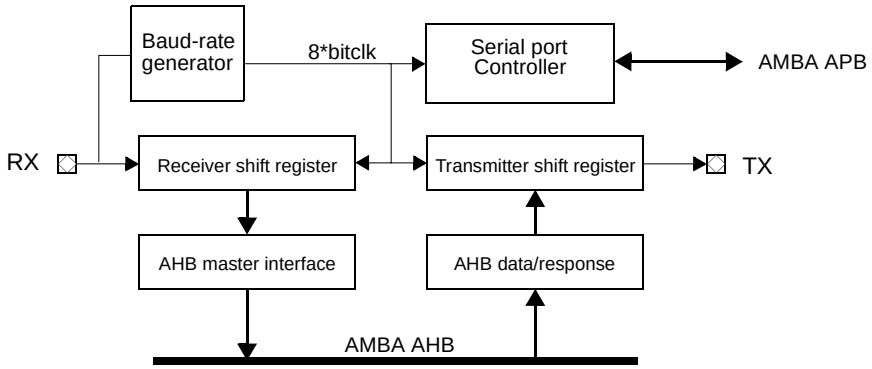


Figure 14. **Block diagram**

12.2 Operation

12.2.1 Transmission protocol

The interface supports a simple protocol where commands consist of a control byte, followed by a 32-bit address, followed by optional write data. Write access does not return any response, while a read access only returns the read data. Data is sent on 8-bit basis as shown below.

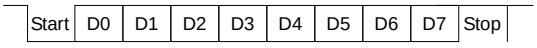


Figure 15. **Data frame**

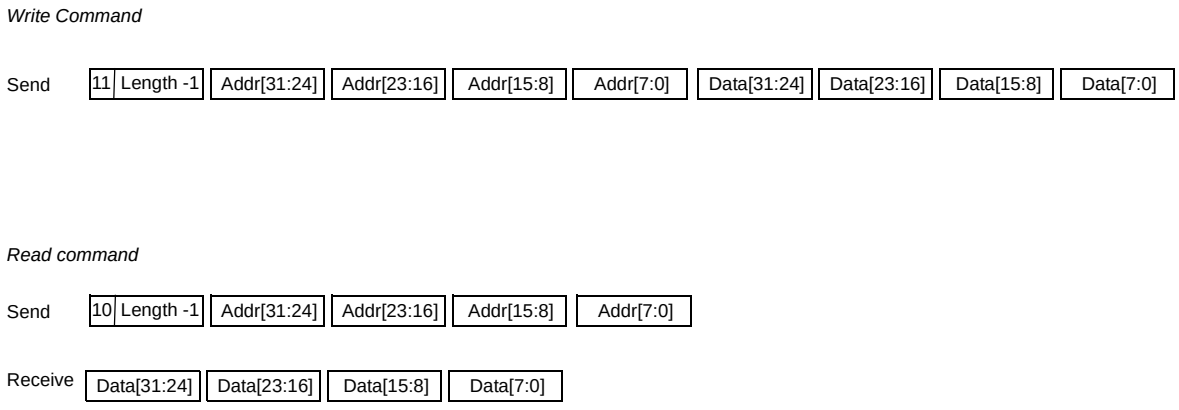


Figure 16. **Commands**

Block transfers can be performed be setting the length field to n-1, where n denotes the number of transferred words. For write accesses, the control byte and address is sent once, followed by the number of data words to be written. The address is automatically incremented after each data word. For

read accesses, the control byte and address is sent once and the corresponding number of data words is returned.

12.2.2 Baud rate generation

The UART contains a 18-bit down-counting scaler to generate the desired baud-rate. The scaler is clocked by the system clock and generates a UART tick each time it underflows. The scaler is reloaded with the value of the UART scaler reload register after each underflow. The resulting UART tick frequency should be 8 times the desired baud-rate.

If not programmed by software, the baud rate will be automatically discovered. This is done by searching for the shortest period between two falling edges of the received data (corresponding to two bit periods). When three identical two-bit periods has been found, the corresponding scaler reload value is latched into the reload register, and the BL bit is set in the UART control register. If the BL bit is reset by software, the baud rate discovery process is restarted. The baud-rate discovery is also restarted when a ‘break’ or framing error is detected by the receiver, allowing to change to baudrate from the external transmitter. For proper baudrate detection, the value 0x55 should be transmitted to the receiver after reset or after sending break.

The best scaler value for manually programming the baudrate can be calculated as follows:

```
scaler = (((system_clk*10)/(baudrate*8))-5)/10
```

12.3 Registers

The core is programmed through registers mapped into APB address space.

Table 55. AHB UART registers

APB address offset	Register
0x4	AHB UART status register
0x8	AHB UART control register
0xC	AHB UART scaler register

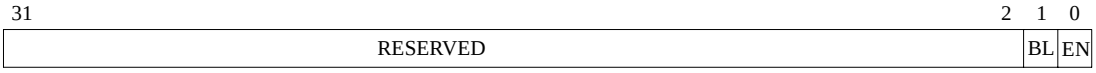


Figure 17. **AHB UART control register**

0: Receiver enable (EN) - if set, enables both the transmitter and receiver. Reset value: ‘0’.
1: Baud rate locked (BL) - is automatically set when the baud rate is locked. Reset value: ‘0’.

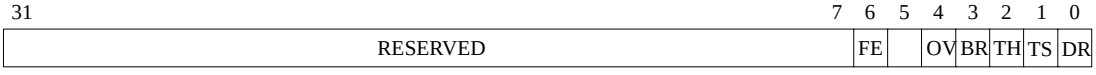


Figure 18. **AHB UART status register**

0: Data ready (DR) - indicates that new data has been received by the AMBA AHB master interface. Read only. Reset value: ‘0’.
1: Transmitter shift register empty (TS) - indicates that the transmitter shift register is empty. Read only. Reset value: ‘1’
2: Transmitter hold register empty (TH) - indicates that the transmitter hold register is empty. Read only. Reset value: ‘1’
3: Break (BR) - indicates that a BREAKE has been received. Reset value: ‘0’
4: Overflow (OV) - indicates that one or more character have been lost due to receiver overflow. Reset value: ‘0’
6: Frame error (FE) - indicates that a framing error was detected. Reset value: ‘0’

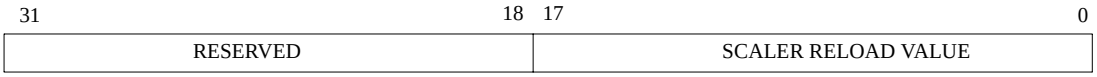


Figure 19. **AHB UART scaler reload register**

17:0 Baudrate scaler reload value = (((system_clk*10)/(baudrate*8))-5)/10. Reset value: “3FFFF“.

12.4 Vendor and device identifiers

The core has vendor identifier 0x01 (Gaisler Research) and device identifier 0x007. For description of vendor and device identifiers see GRLIB IP Library User’s Manual.

12.5 Configuration options

Table 56 shows the configuration options of the core (VHDL generics).

Table 56. Configuration options

Generic	Function	Allowed range	Default
hindex	AHB master index	0 - NAHBMST-1	0
pindex	APB slave index	0 - NAPBSLV-1	0
paddr	ADDR field of the APB BAR.	0 - 16#FFF#	0
pmask	MASK field of the APB BAR.	0 - 16#FFF#	16#FFF#

12.6 Signal descriptions

Table 57 shows the interface signals of the core (VHDL ports)..

Table 57. Signal descriptions

Signal name	Field	Type	Function	Active
RST	N/A	Input	Reset	Low
CLK	N/A	Input	Clock	-
UARTI	RXD	Input	UART receiver data	High
	CTSN	Input	UART clear-to-send	High
	EXTCLK	Input	Use as alternative UART clock	-
UARTO	RTSN	Output	UART request-to-send	High
	TXD	Output	UART transmit data	High
APBI	*	Input	APB slave input signals	-
APBO	*	Output	APB slave output signals	-
AHBI	*	Input	AMB master input signals	-
AHBO	*	Output	AHB master output signals	-

* see GRLIB IP Library User’s Manual

12.7 Library dependencies

Table 58 shows libraries used when instantiating the core (VHDL libraries).

Table 58. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AMBA signal definitions
GAISLER	UART	Signals, component	Signals and component declaration

12.8 Instantiation

This example shows how the core can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
library gaisler;
use gaisler.uart.all;

entity ahbuart_ex is
  port (
    clk : in std_ulogic;
    rstn : in std_ulogic;

    -- UART signals
    ahbrxd   : in  std_ulogic;
    ahbtxd   : out std_ulogic
    );
end;

architecture rtl of ahbuart_ex is

  -- AMBA signals
  signal apbi  : apb_slv_in_type;
  signal apbo  : apb_slv_out_vector := (others => apb_none);
  signal ahbmi : ahb_mst_in_type;
  signal ahbmo : ahb_mst_out_vector := (others => ahbm_none);

  -- UART signals
  signal ahbuarti : uart_in_type;
  signal ahbuarto : uart_out_type;

begin

  -- AMBA Components are instantiated here
  ...


  -- AHB UART
  ahbuart0 : ahbuart
  generic map (hindex => 5, pindex => 7, paddr => 7)
  port map (rstn, clk, ahbuarti, ahbuarto, apbi, apbo(7), ahbmi, ahbmo(5));

  -- AHB UART input data
  ahbuarti.rxd <= ahbrxd;

  -- connect AHB UART output to entity output signal
  ahbtxd <= ahbuarto.txd;

end;
```

Beilage 10

APBUART GRLIB Auszug

16 APBUART - AMBA APB UART Serial Interface

16.1 Overview

The interface is provided for serial communications. The UART supports data frames with 8 data bits, one optional parity bit and one stop bit. To generate the bit-rate, each UART has a programmable 12-bit clock divider. Two FIFOs are used for data transfer between the APB bus and UART, when *fifosize* VHDL generic > 1. Two holding registers are used data transfer between the APB bus and UART, when *fifosize* VHDL generic = 1. Hardware flow-control is supported through the RTSN/CTSN handshake signals, when *flow* VHDL generic is set. Parity is supported, when *parity* VHDL generic is set.

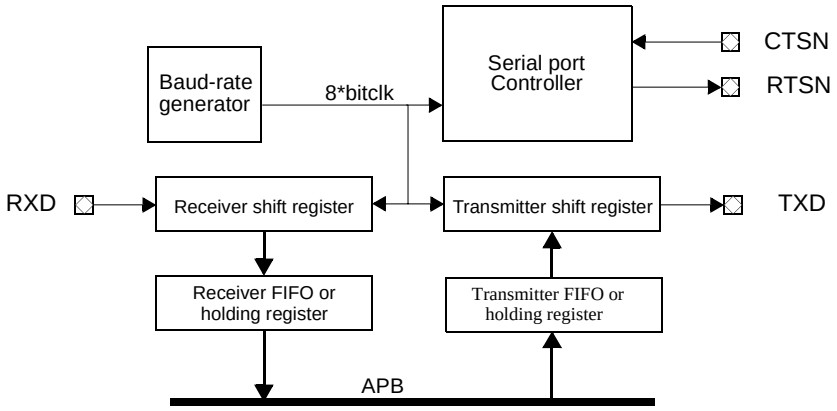


Figure 30. Block diagram

16.2 Operation

16.2.1 Transmitter operation

The transmitter is enabled through the TE bit in the UART control register. Data that is to be transferred is stored in the FIFO/holding register by writing to the data register. This FIFO is configurable to different sizes via the *fifosize* VHDL generic. When the size is 1, only a single holding register is used but in the following discussion both will be referred to as FIFOs. When ready to transmit, data is transferred from the transmitter FIFO/holding register to the transmitter shift register and converted to a serial stream on the transmitter serial output pin (TXD). It automatically sends a start bit followed by eight data bits, an optional parity bit, and one stop bit (figure 31). The least significant bit of the data is sent first.

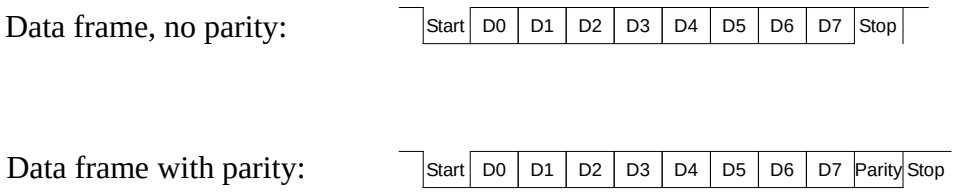


Figure 31. UART data frames

Following the transmission of the stop bit, if a new character is not available in the transmitter FIFO, the transmitter serial data output remains high and the transmitter shift register empty bit (TS) will be set in the UART status register. Transmission resumes and the TS is cleared when a new character is loaded into the transmitter FIFO. When the FIFO is empty the TE bit is set in the status register. If the transmitter is disabled, it will immediately stop any active transmissions including the character currently being shifted out from the transmitter shift register. The transmitter holding register may not be loaded when the transmitter is disabled or when the FIFO (or holding register) is full. If this is done, data might be overwritten and one or more frames are lost.

The discussion above applies to any FIFO configurations including the special case with a holding register (VHDL generic *fifosize* = 1). If FIFOs are used (VHDL generic *fifosize* > 1) some additional status and control bits are available. The TF status bit (not to be confused with the TF control bit) is set if the transmitter FIFO is currently full and the TH bit is set as long as the FIFO is *less* than half-full (less than half of entries in the FIFO contain data). The TF control bit enables FIFO interrupts when set. The status register also contains a counter (TCNT) showing the current number of data entries in the FIFO.

When flow control is enabled, the CTSN input must be low in order for the character to be transmitted. If it is deasserted in the middle of a transmission, the character in the shift register is transmitted and the transmitter serial output then remains inactive until CTSN is asserted again. If the CTSN is connected to a receivers RTSN, overrun can effectively be prevented.

16.2.2 Receiver operation

The receiver is enabled for data reception through the receiver enable (RE) bit in the UART control register. The receiver looks for a high to low transition of a start bit on the receiver serial data input pin. If a transition is detected, the state of the serial input is sampled a half bit clocks later. If the serial input is sampled high the start bit is invalid and the search for a valid start bit continues. If the serial input is still low, a valid start bit is assumed and the receiver continues to sample the serial input at one bit time intervals (at the theoretical centre of the bit) until the proper number of data bits and the parity bit have been assembled and one stop bit has been detected. The serial input is shifted through an 8-bit shift register where all bits have to have the same value before the new value is taken into account, effectively forming a low-pass filter with a cut-off frequency of 1/8 system clock.

The receiver also has a configurable FIFO which is identical to the one in the transmitter. As mentioned in the transmitter part, both the holding register and FIFO will be referred to as FIFO.

During reception, the least significant bit is received first. The data is then transferred to the receiver FIFO and the data ready (DR) bit is set in the UART status register as soon as the FIFO contains at least one data frame. The parity, framing and overrun error bits are set at the received byte boundary, at the same time as the receiver ready bit is set. The data frame is not stored in the FIFO if an error is detected. Also, the new error status bits are or:ed with the old values before they are stored into the status register. Thus, they are not cleared until written to with zeros from the AMBA APB bus. If both

the receiver FIFO and shift registers are full when a new start bit is detected, then the character held in the receiver shift register will be lost and the overrun bit will be set in the UART status register.

If flow control is enabled, then the RTSN will be negated (high) when a valid start bit is detected and the receiver FIFO is full. When the holding register is read, the RTSN will automatically be reasserted again.

When the VHDL generic *fifosize* > 1, which means that holding registers are not considered here, some additional status and control bits are available. The RF status bit (not to be confused with the RF control bit) is set when the receiver FIFO is full. The RH status bit is set when the receiver FIFO is half-full (at least half of the entries in the FIFO contain data frames). The RF control bit enables receiver FIFO interrupts when set. A RCNT field is also available showing the current number of data frames in the FIFO.

16.3 Baud-rate generation

Each UART contains a 12-bit down-counting scaler to generate the desired baud-rate. The scaler is clocked by the system clock and generates a UART tick each time it underflows. It is reloaded with the value of the UART scaler reload register after each underflow. The resulting UART tick frequency should be 8 times the desired baud-rate. If the EC bit is set, the scaler will be clocked by the external clock input rather than the system clock. In this case, the frequency of external clock must be less than half the frequency of the system clock.

16.4 Loop back mode

If the LB bit in the UART control register is set, the UART will be in loop back mode. In this mode, the transmitter output is internally connected to the receiver input and the RTSN is connected to the CTSN. It is then possible to perform loop back tests to verify operation of receiver, transmitter and associated software routines. In this mode, the outputs remain in the inactive state, in order to avoid sending out data.

16.5 FIFO debug mode

FIFO debug mode is entered by setting the debug mode bit in the control register. In this mode it is possible to read the transmitter FIFO and write the receiver FIFO through the FIFO debug register. The transmitter output is held inactive when in debug mode. A write to the receiver FIFO generates an interrupt if receiver interrupts are enabled.

16.6 Interrupt generation

Interrupts are generated differently when a holding register is used (VHDL generic *fifosize* = 1) and when FIFOs are used (VHDL generic *fifosize* > 1). When holding registers are used, the UART will generate an interrupt under the following conditions: when the transmitter is enabled, the transmitter interrupt is enabled and the transmitter holding register moves from full to empty; when the receiver is enabled, the receiver interrupt is enabled and the receiver holding register moves from empty to full; when the receiver is enabled, the receiver interrupt is enabled and a character with either parity, framing or overrun error is received.

For FIFOs, two different kinds of interrupts are available: normal interrupts and FIFO interrupts. For the transmitter, normal interrupts are generated when transmitter interrupts are enabled (TI), the transmitter is enabled and the transmitter FIFO goes from containing data to being empty. FIFO interrupts are generated when the FIFO interrupts are enabled (TF), transmissions are enabled (TE) and the UART is less than half-full (that is, whenever the TH status bit is set). This is a level interrupt and the interrupt signal is continuously driven high as long as the condition prevails. The receiver interrupts work in the same way. Normal interrupts are generated in the same manner as for the holding register. FIFO interrupts are generated when receiver FIFO interrupts are enabled, the receiver is enabled and

the FIFO is half-full. The interrupt signal is continuously driven high as long as the receiver FIFO is half-full (at least half of the entries contain data frames).

16.7 Registers

The core is controlled through registers mapped into APB address space.

Table 82. UART registers

APB address offset	Register
0x0	UART Data register
0x4	UART Status register
0x8	UART Control register
0xC	UART Scaler register
0x10	UART FIFO debug register

16.7.1 UART Data Register

Table 83. UART data register

31	8	7	0
RESERVED		DATA	

7: 0 Receiver holding register or FIFO (read access)

7: 0 Transmitter holding register or FIFO (write access)

16.7.2 UART Status Register

Table 84. UART status register

31	26	25	20	19	11	10	9	8	7	6	5	4	3	2	1	0
RCNT		TCNT		RESERVED		RF	TF	RH	TH	FE	PE	OV	BR	TE	TS	DR

31: 26 Receiver FIFO count (RCNT) - shows the number of data frames in the receiver FIFO.

25: 20 Transmitter FIFO count (TCNT) - shows the number of data frames in the transmitter FIFO.

10 Receiver FIFO full (RF) - indicates that the Receiver FIFO is full.

9 Transmitter FIFO full (TF) - indicates that the Transmitter FIFO is full.

8 Receiver FIFO half-full (RH) -indicates that at least half of the FIFO is holding data.

7 Transmitter FIFO half-full (TH) - indicates that the FIFO is less than half-full.

6 Framing error (FE) - indicates that a framing error was detected.

5 Parity error (PE) - indicates that a parity error was detected.

4 Overrun (OV) - indicates that one or more character have been lost due to overrun.

3 Break received (BR) - indicates that a BREAK has been received.

2 Transmitter FIFO empty (TE) - indicates that the transmitter FIFO is empty.

1 Transmitter shift register empty (TS) - indicates that the transmitter shift register is empty.

0 Data ready (DR) - indicates that new data is available in the receiver holding register

16.7.3 UART Control Register

Table 85. UART control register

31	30	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FA	RESERVED		OE	DB	RF	TF	EC	LB	FL	PE	PS	TI	RI	TE	RE

31	FIFOs available (FA) - Set to 1 when receiver and transmitter FIFOs are available. When 0, only holding register are available.
30: 13	RESERVED
12	Output enable (OE) - Transmitter output enable. Driven directly on the txen output.
11	FIFO debug mode enable (DB) - when set, it is possible to read and write the FIFO debug register
10	Receiver FIFO interrupt enable (RF) - when set, Receiver FIFO level interrupts are enabled
9	Transmitter FIFO interrupt enable (TF) - when set, Transmitter FIFO level interrupts are enabled.
8	External Clock (EC) - if set, the UART scaler will be clocked by UARTI.EXTCLK
7	Loop back (LB) - if set, loop back mode will be enabled
6	Flow control (FL) - if set, enables flow control using CTS/RTS (when implemented)
5	Parity enable (PE) - if set, enables parity generation and checking (when implemented)
4	Parity select (PS) - selects parity polarity (0 = even parity, 1 = odd parity) (when implemented)
3	Transmitter interrupt enable (TI) - if set, interrupts are generated when a frame is transmitted
2	Receiver interrupt enable (RI) - if set, interrupts are generated when a frame is received
1	Transmitter enable (TE) - if set, enables the transmitter.
0	Receiver enable (RE) - if set, enables the receiver.

16.7.4 UART Scaler Register

Table 86. UART scaler reload register

31	12	11	0
RESERVED		SCALER RELOAD VALUE	

11: 0	Scaler reload value

16.7.5 UART FIFO Debug Register

Table 87. UART FIFO debug register

31	8	7	0
RESERVED		DATA	

7: 0	Transmitter holding register or FIFO (read access)
7: 0	Receiver holding register or FIFO (write access)

16.8 Vendor and device identifiers

The core has vendor identifier 0x01 (Gaisler Research) and device identifier 0x00C. For a description of vendor and device identifiers see GRLIB IP Library User’s Manual.

16.9 Configuration options

Table 88 shows the configuration options of the core (VHDL generics).

Table 88. Configuration options

Generic	Function	Allowed range	Default
pindex	APB slave index	0 - NAPBSLV-1	0
paddr	ADDR field of the APB BAR.	0 - 16#FFF#	0
pmask	MASK field of the APB BAR.	0 - 16#FFF#	16#FFF#
console	Prints output from the UART on console during VHDL simulation and speeds up simulation by always returning '1' for Data Ready bit of UART Status register. Does not effect synthesis.	0 - 1	0
pirq	Index of the interrupt line.	0 - NAHBIRQ-1	0
parity	Enables parity	0 - 1	1
flow	Enables flow control	0 - 1	1
fifosize	Selects the size of the Receiver and Transmitter FIFOs	1, 2, 4, 8, 16, 32	1
abits	Select the number of APB address bits used to decode the register addresses	3 - 8	8

16.10 Signal descriptions

Table 89 shows the interface signals of the core (VHDL ports).

Table 89. Signal descriptions

Signal name	Field	Type	Function	Active
RST	N/A	Input	Reset	Low
CLK	N/A	Input	Clock	-
APBI	*	Input	APB slave input signals	-
APBO	*	Output	APB slave output signals	-
UARTI	RXD	Input	UART receiver data	-
	CTSN	Input	UART clear-to-send	Low
	EXTCLK	Input	Use as alternative UART clock	-
UARTO	RTSN	Output	UART request-to-send	Low
	TXD	Output	UART transmit data	-
	SCALER	Output	UART scaler value	-
	TXEN	Output	Output enable for transmitter	High
	FLOW	Output	Unused	-
	RXEN	Output	Receiver enable	High

* see GRLIB IP Library User's Manual

16.11 Library dependencies

Table 90 shows libraries that should be used when instantiating the core.

Table 90. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	APB signal definitions
GAISLER	UART	Signals, component	Signal and component declaration

16.12 Instantiation

This example shows how the core can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
library gaisler;
use gaisler.uart.all;

entity apbuart_ex is
  port (
    clk : in std_ulogic;
    rstn : in std_ulogic;

    -- UART signals
    rxd   : in  std_ulogic;
    txd   : out std_ulogic
    );
end;

architecture rtl of apbuart_ex is

  -- APB signals
  signal apbi  : apb_slv_in_type;
  signal apbo  : apb_slv_out_vector := (others => apb_none);

  -- UART signals
  signal uarti : uart_in_type;
  signal uarto : uart_out_type;

begin

  -- AMBA Components are instantiated here
   ...


  -- APB UART
  uart0 : apbuart
  generic map (pindex => 1, paddr => 1,  pirq => 2,
console => 1, fifosize => 1)
  port map (rstn, clk, apbi, apbo(1), uarti, uarto);

  -- UART input data
  uarti.rxd <= rxd;

  -- APB UART inputs not used in this configuration
  uarti.ctsn <= '0'; uarti.extclk <= '0';

  -- connect APB UART output to entity output signal
  txd <= uarto.txd;

end;
```

Beilage 11

IRQMP GRLIB Auszug

54 IRQMP - Multiprocessor Interrupt Controller

54.1 Overview

The AMBA system in GRLIB provides an interrupt scheme where interrupt lines are routed together with the remaining AHB/APB bus signals, forming an interrupt bus. Interrupts from AHB and APB units are routed through the bus, combined together, and propagated back to all units. The multiprocessor interrupt controller core is attached to AMBA bus as an APB slave, and monitors the combined interrupt signals.

The interrupts generated on the interrupt bus are all forwarded to the interrupt controller. The interrupt controller prioritizes, masks and propagates the interrupt with the highest priority to the processor. In multiprocessor systems, the interrupts are propagated to all processors.

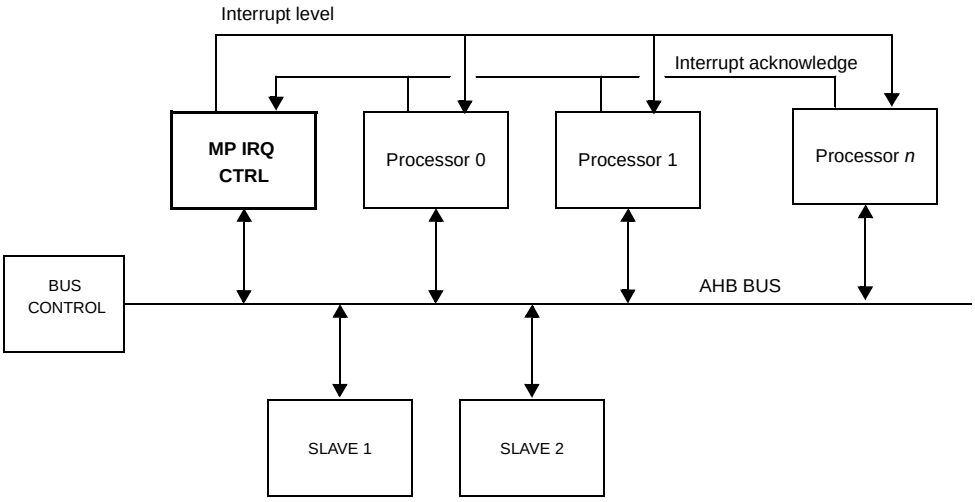


Figure 162. LEON3 multiprocessor system with Multiprocessor Interrupt controller

54.2 Operation

54.2.1 Interrupt prioritization

The interrupt controller monitors interrupt 1 - 15 of the interrupt bus (APBI.PIRQ[15:1]). When any of these lines are asserted high, the corresponding bit in the interrupt pending register is set. The pending bits will stay set even if the PIRQ line is de-asserted, until cleared by software or by an interrupt acknowledge from the processor.

Each interrupt can be assigned to one of two levels (0 or 1) as programmed in the interrupt level register. Level 1 has higher priority than level 0. The interrupts are prioritised within each level, with interrupt 15 having the highest priority and interrupt 1 the lowest. The highest interrupt from level 1 will be forwarded to the processor. If no unmasked pending interrupt exists on level 1, then the highest unmasked interrupt from level 0 will be forwarded. PIRQ[31:16] are not used by the IRQMP core.

Interrupts are prioritised at system level, while masking and forwarding of interrupts in done for each processor separately. Each processor in an multiprocessor system has separate interrupt mask and force registers. When an interrupt is signalled on the interrupt bus, the interrupt controller will prioritize interrupts, perform interrupt masking for each processor according to the mask in the corresponding mask register and forward the interrupts to the processors.

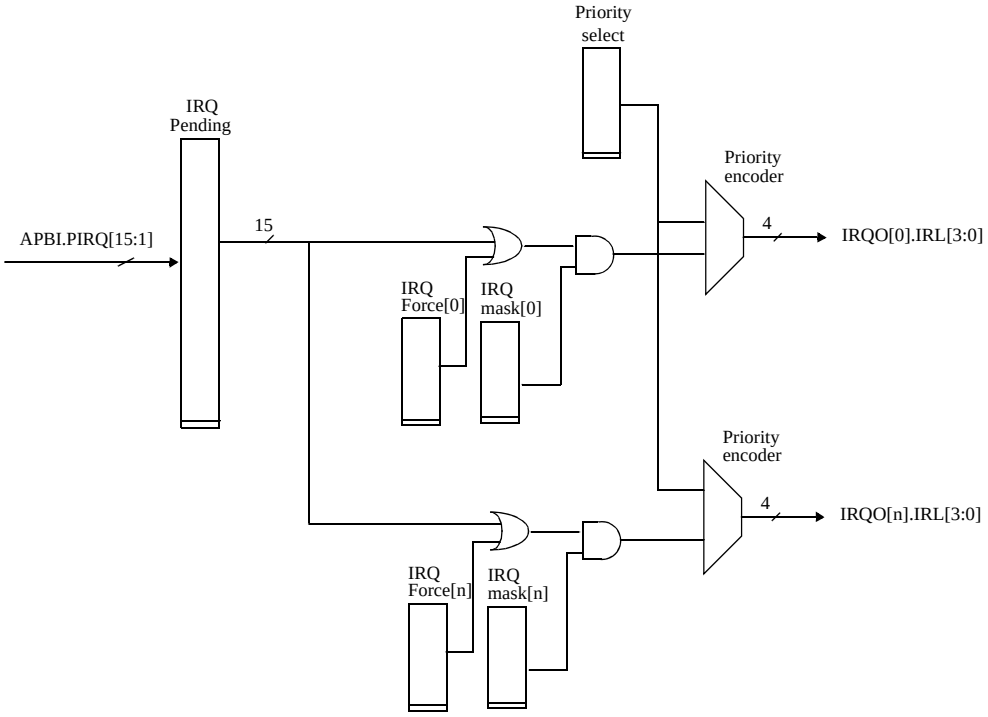


Figure 163. Interrupt controller block diagram

When a processor acknowledges the interrupt, the corresponding pending bit will automatically be cleared. Interrupt can also be forced by setting a bit in the interrupt force register. In this case, the processor acknowledgement will clear the force bit rather than the pending bit. After reset, the interrupt mask register is set to all zeros while the remaining control registers are undefined. Note that interrupt 15 cannot be maskable by the LEON3 processor and should be used with care - most operating systems do not safely handle this interrupt.

54.2.2 Processor status monitoring

The processor status can be monitored through the Multiprocessor Status Register. The STATUS field in this register indicates if a processor is halted (‘1’) or running (‘0’). A halted processor can be reset and restarted by writing a ‘1’ to its status field. After reset, all processors except processor 0 are halted. When the system is properly initialized, processor 0 can start the remaining processors by writing to their STATUS bits.

54.2.3 Irq broadcasting

The Broadcast Register is activated when the generic *ncpu* is > 1. A incoming irq that has its bit set in the Broadcast Register is propagated to the force register of *all* CPUs rather than only to the Pending Register. This can be used to implement a timer that fires to all cpus with that same irq.

54.3 Registers

The core is controlled through registers mapped into APB address space. The number of implemented registers depend on number of processor in the multiprocessor system.

*Table 619.*Interrupt Controller registers

APB address offset	Register
0x00	Interrupt level register
0x04	Interrupt pending register
0x08	Interrupt force register (NCPU = 0)
0x0C	Interrupt clear register
0x10	Multiprocessor status register
0x14	Broadcast register
0x40	Processor interrupt mask register
0x44	Processor 1 interrupt mask register
0x40 + 4 * *n*	Processor *n* interrupt mask register
0x80	Processor interrupt force register
0x84	Processor 1 interrupt force register
0x80 + 4 * *n*	Processor *n* interrupt force register
0xC4	Processor 1 extended interrupt identification register
0xC0 + 4 * *n*	Processor *n* extended interrupt identification register

54.3.1 Interrupt level register

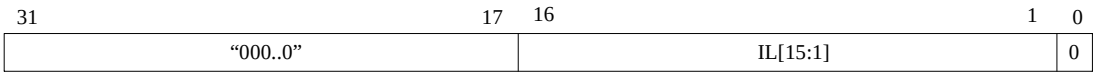


Figure 164. Interrupt level register

[31:16] Reserved.
[15:1] Interrupt Level *n* (IL[*n*]): Interrupt level for interrupt *n*.
[0] Reserved.

54.3.2 Interrupt pending register

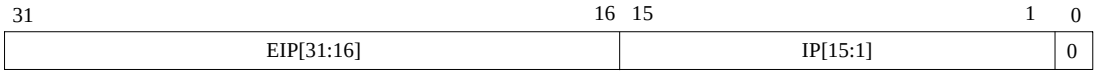


Figure 165. Interrupt pending register

[31:17] Extended Interrupt Pending n (EIP[n]).
[15:1] Interrupt Pending n (IP[n]): Interrupt pending for interrupt n.
[0] Reserved

54.3.3 Interrupt force register (NCPU = 0)

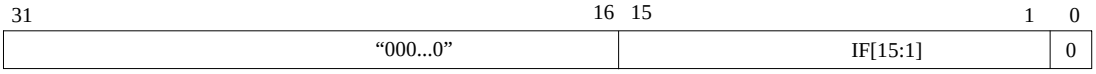


Figure 166. Interrupt force register

[31:16] Reserved.
[15:1] Interrupt Force *n* (IF[*n*]): Force interrupt nr *n*.
[0] Reserved.

54.3.4 Interrupt clear register

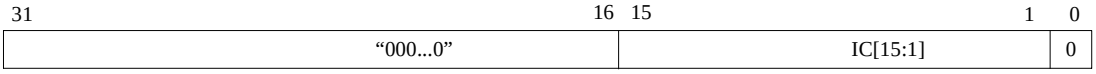


Figure 167. Interrupt clear register

[31:16] Reserved.
[15:1] Interrupt Clear *n* (IC[*n*]): Writing ‘1’ to IC*n* will clear interrupt *n*.
[0] Reserved.

54.3.5 Multiprocessor status register

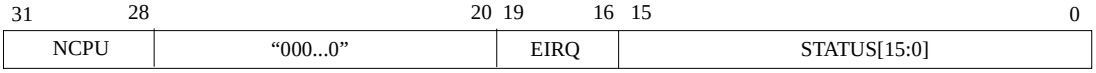


Figure 168. Multiprocessor status register

[31:28] NCPU. Number of CPU’s in the system -1 .
[19:16] EIRQ. Interrupt number (1 - 15) used for extended interrupts. Fixed to 0 if extended interrupts are disabled.
[15:1] Power-down status of CPU [*n*]: ‘1’ = power-down, ‘0’ = running. Write with ‘1’ to start processor *n.*

54.3.6 Processor interrupt mask register

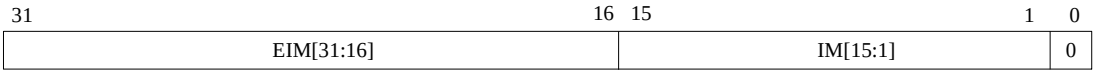


Figure 169. Processor interrupt mask register

[31:16] Interrupt mask for extended interrupts
[15:1] Interrupt Mask *n* (IM[*n*]): If IMn = 0 the interrupt *n* is masked, otherwise it is enabled.
[0] Reserved.

54.3.7 Broadcast register (NCPU > 1)

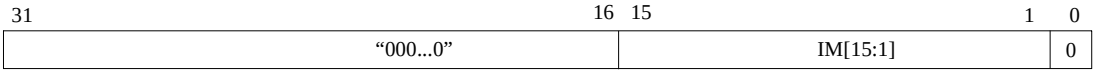


Figure 170. Processor interrupt mask register

[31:16] Reserved.
[15:1] Broadcast Mask *n* (BM[*n*]): If BMn = 1 the interrupt *n* is broadcasted (written to the Force Register of all CPUs), otherwise standard semantic applies (Pending Register).
[0] Reserved.

54.3.8 Processor interrupt force register (NCPU > 0)

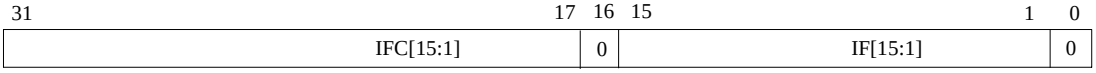


Figure 171. Processor interrupt force register

[31:17] Interrupt force clear *n* (IFC[*n*]).
[15:1] Interrupt Force *n* (IF[*n*]): Force interrupt nr *n*.
[0] Reserved.

54.3.9 Extended interrupt identification register

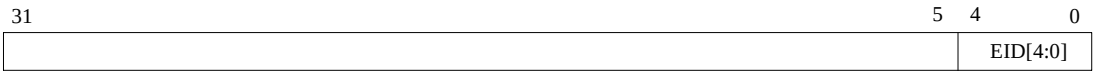


Figure 172. Processor interrupt force register

[4:0] ID (16 - 31) of the acknowledged extended interrupt

54.4 Vendor and device identifiers

The core has vendor identifier 0x01 (Gaisler Research) and device identifier 0x00D. For description of vendor and device identifiers see GRLIB IP Library User’s Manual.

54.5 Configuration options

Table 620 shows the configuration options of the core (VHDL generics).

*Table 620.*Configuration options

Generic	Function	Allowed range	Default
pindex	Selects which APB select signal (PSEL) will be used to access the timer unit	0 to NAPBMAX-1	0
paddr	The 12-bit MSB APB address	0 to 4095	0
pmask	The APB address mask	0 to 4095	4095
ncpu	Number of processors in multiprocessor system	1 to 16	1

54.6 Signal descriptions

Table 621 shows the interface signals of the core (VHDL ports).

*Table 621.*Signal descriptions

Signal name	Field	Type	Function	Active
RST	N/A	Input	Reset	Low
CLK	N/A	Input	Clock	-
APBI	*	Input	APB slave input signals	-
APBO	*	Output	APB slave output signals	-
IRQI[n]	INTACK	Input	Processor *n* Interrupt acknowledge	High
	IRL[3:0]		Processor *n* interrupt level	High
IRQO[n]	IRL[3:0]	Output	Processor *n* Input interrupt level	High
	RST		Reset power-down and error mode of processor *n*	High
	RUN		Start processor *n* after reset (SMP systems only)	High

* see GRLIB IP Library User's Manual

54.7 Library dependencies

Table 622 shows libraries that should be used when instantiating the core.

*Table 622.*Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AMBA signal definitions
GAISLER	LEON3	Signals, component	Signals and component declaration

54.8 Instantiation

This example shows how the core can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
library gaisler;
use gaisler.leon3.all;

entity irqmp_ex is
  port (
    clk : in std_ulogic;
    rstn : in std_ulogic;

    ...  -- other signals
    );
end;

architecture rtl of irqmp_ex is
  constant NCPU : integer := 4;

  -- AMBA signals
  signal apbi  : apb_slv_in_type;
  signal apbo  : apb_slv_out_vector := (others => apb_none);
  signal ahbmi : ahb_mst_in_type;
  signal ahbmo : ahb_mst_out_vector := (others => ahbm_none);
  signal ahbsi : ahb_slv_in_type;
```

```
  -- GP Timer Unit input signals
  signal irqi   : irq_in_vector(0 to NCPU-1);
  signal irqo   : irq_out_vector(0 to NCPU-1);

  -- LEON3 signals
  signal leon3i : l3_in_vector(0 to NCPU-1);
  signal leon3o : l3_out_vector(0 to NCPU-1);

begin

  -- 4 LEON3 processors are instantiated here
  cpu : for i in 0 to NCPU-1 generate
    u0 : leon3s generic map (hindex => i)
    port map (clk, rstn, ahbmi, ahbmo(i), ahbsi,
irqi(i), irqo(i), dbgi(i), dbgo(i));
  end generate;

  -- MP IRQ controller
  irqctrl0 : irqmp
  generic map (pindex => 2, paddr => 2, ncpu => NCPU)
  port map (rstn, clk, apbi, apbo(2), irqi, irqo);
end
```

Beilage 12

GRGPIO GRLIB Auszug

47 GRGPIO - General Purpose I/O Port

47.1 Overview

The general purpose input output port core is a scalable and provides optional interrupt support. The port width can be set to 2 - 32 bits through the *nbits* VHDL generic (i.e. *nbits* = 16). Interrupt generation and shaping is only available for those I/O lines where the corresponding bit in the *imask* VHDL generic has been set to 1.

Each bit in the general purpose input output port can be individually set to input or output, and can optionally generate an interrupt. For interrupt generation, the input can be filtered for polarity and level/edge detection.

It is possible to share GPIO pins with other signals. The output register can then be bypassed through the bypass register.

The figure 143 shows a diagram for one I/O line.

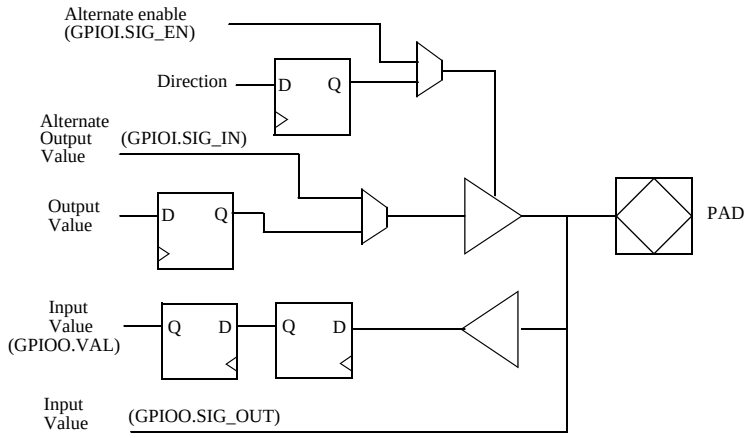


Figure 143. General Purpose I/O Port diagram

47.2 Operation

The I/O ports are implemented as bi-directional buffers with programmable output enable. The input from each buffer is synchronized by two flip-flops in series to remove potential meta-stability. The synchronized values can be read-out from the I/O port data register. They are also available on the GPIOO.VAL signals. The output enable is controlled by the I/O port direction register. A ‘1’ in a bit position will enable the output buffer for the corresponding I/O line. The output value driven is taken from the I/O port output register.

Each I/O port can drive a separate interrupt line on the APB interrupt bus. The interrupt number is equal to the I/O line index (PIO[1] = interrupt 1, etc.). The interrupt generation is controlled by three registers: interrupt mask, polarity and edge registers. To enable an interrupt, the corresponding bit in the interrupt mask register must be set. If the edge register is ‘0’, the interrupt is treated as level sensitive. If the polarity register is ‘0’, the interrupt is active low. If the polarity register is ‘1’, the interrupt is active high. If the edge register is ‘1’, the interrupt is edge-triggered. The polarity register then selects between rising edge (‘1’) or falling edge (‘0’).

A GPIO pin can be shared with other signals. The ports that should have the capability to be shared are specified with the *bypass* generic (the corresponding bit in the generic must be 1). The unfiltered inputs are available through GPIOO.SIG_OUT and the alternate output value must be provided in GPIOI.SIG_IN. The bypass register then controls whether the alternate output is chosen. The direction of the GPIO pin can also be shared, if the correspondig bit is set in the *bpdir* generic. In such case, the output buffer is enabled when GPIOI.SIG_EN is active.

47.3 Registers

The core is programmed through registers mapped into APB address space.

Table 419. General Purpose I/O Port registers

APB address offset	Register
0x00	I/O port data register
0x04	I/O port output register
0x08	I/O port direction register
0x0C	Interrupt mask register
0x10	Interrupt polarity register
0x14	Interrupt edge register
0x18	Bypass register

Table 420. I/O port data register

31	16	16-1	0
“000..0”		I/O port input value	

16-1: 0 I/O port input value

Table 421. I/O port output register

31	16	16-1	0
“000..0”		I/O port output value	

16-1: 0 I/O port output value

Table 422. I/O port direction register

31	16	16-1	0
“000..0”		I/O port direction value	

16-1: 0 I/O port direction value (0=output disabled, 1=output enabled)

Table 423. Interrupt mask register

31	16	16-1	0
“000..0”		Interrupt mask	

16-1: 0 Interrupt mask (0=interrupt masked, 1=intrrupt enabled)

Table 424. Interrupt polarity register

31	16	16-1	0
“000..0”		Interrupt polarity	

16-1: 0 Interrupt polarity (0=low/falling, 1=high/rising)

Table 425. Interrupt edge register

31	16	16-1	0
“000..0”		Interrupt edge	

16-1: 0 Interrupt edge (0=level, 1=edge)

Table 426. Bypass register

31	16	16-1	0
“000..0”		Bypass	

Table 426. Bypass register

16-1: 0 Bypass. (0=normal output, 1=alternate output)

47.4 Vendor and device identifiers

The core has vendor identifier 0x01 (Gaisler Research) and device identifier 0x01A. For description of vendor and device identifiers see GRLIB IP Library User's Manual.

47.5 Configuration options

Table 427 shows the configuration options of the core (VHDL generics).

*Table 427.*Configuration options

Generic	Function	Allowed range	Default
pindex	Selects which APB select signal (PSEL) will be used to access the GPIO unit	0 to NAPBMAX-1	0
paddr	The 12-bit MSB APB address	0 to 16#FFF#	0
pmask	The APB address mask	0 to 16#FFF#	16#FFF#
nbits	Defines the number of bits in the I/O port	1 to 32	8
imask	Defines which I/O lines are provided with interrupt generation and shaping	0 - 16#FFFF#	0
oepol	Select polarity of output enable signals. 0 = active low, 1 = active high.	0 - 1	0
syncrst	Selects between syncronous (1) or asyncronous (0) reset during power-up.	0 - 1	0
bypass	Defines which I/O lines are provided bypass capabilities	0 - 16#7FFFFFFF#	0
scantest	Enable scan support for asyncronous-reset flip-flops	0 - 1	0
bpdir	Defines which I/O lines are provided output enable bypass capabilities	0 - 16#7FFFFFFF#	0

47.6 Signal descriptions

Table 428 shows the interface signals of the core (VHDL ports).

*Table 428.*Signal descriptions

Signal name	Field	Type	Function	Active
RST	N/A	Input	Reset	Low
CLK	N/A	Input	Clock	-
APBI	*	Input	APB slave input signals	-
APBO	*	Output	APB slave output signals	-
GPIOO	OEN[31:0]	Output	I/O port output enable	see oepol
	DOUT[31:0]	Output	I/O port outputs	-
	VAL[31:0]	Output	The current (synchronized) value of the GPIO signals	-
	SIG_OUT[31:0]	Output	The current (unsynchronized) value of the GPIO signals	
GPIOI	DIN[31:0]	Input	I/O port inputs	-
	SIG_IN[31:0]	Input	Alternate output	-

* see GRLIB IP Library User's Manual

47.7 Library dependencies

Table 429 shows libraries used when instantiating the core (VHDL libraries).

*Table 429.*Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AMBA signal definitions
GAISLER	MISC	Signals, component	Component declaration

47.8 Component declaration

The core has the following component declaration.

```
ibrary gaisler;
use gaisler.misc.all;

entity grgpio is
  generic (
    pindex   : integer := 0;
    paddr    : integer := 0;
    pmask    : integer := 16#fff#;
    imask    : integer := 16#0000#;
    nbits    : integer := 16-- GPIO bits

  );
  port (
    rst    : in  std_ulogic;
    clk    : in  std_ulogic;
    apbi   : in  apb_slv_in_type;
    apbo   : out apb_slv_out_type;
    gpioi  : in  gpio_in_type;
    gpioo  : out gpio_out_type
  );
end;
```

47.9 Instantiation

This example shows how the core can be instantiated.

```
library grlib;
use grlib.amba.all;
library gaisler;
use gaisler.misc.all;

signal gpti : gptimer_in_type;

begin

gpio0 : if CFG_GRGPIO_EN /= 0 generate     -- GR GPIO unit
    grgpio0: grgpio
      generic map( pindex => 11, paddr => 11, imask => CFG_GRGPIO_IMASK, nbits => 8)
      port map( rstn, clkm, apbi, apbo(11), gpioi, gpioo);

      pio_pads : for i in 0 to 7 generate
        pio_pad : iopad generic map (tech => padtech)
            port map (gpio(i), gpioo.dout(i), gpioo.oen(i), gpioi.din(i));
      end generate;
end generate;
```

Beilage 13

config.vhd - VHDL Generics

```
--------------------------------------------------------------------------------
-- Generated by HES-SO\\VS' AMBAdraw 1.0.16-b2314_2_2007.1
--   @ Tue May 12 09:17:41 2009
--------------------------------------------------------------------------------
--
--------------------------------------------------------------------------------
-- Copyright 2008, 2009, ISYS, HES-SO//Valais Wallis
--
-- This program is free software: you can redistribute it and/or modify
-- it under the terms of the GNU General Public License as published by
-- the Free Software Foundation; either version 3 of the License, or
-- (at your option) any later version.
--
-- This program IS distributed in the hope that it will be useful,
-- but WITHOUT ANY WARRANTY; without even the implied warranty of
-- MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
-- GNU General Public License for more details.
-- You should have received a copy of the GNU General Public License along with
-- this program. If not, see <http://www.gnu.org/licenses/>.
--------------------------------------------------------------------------------
--

LIBRARY ieee;
    USE ieee.std_logic_1164.ALL;
LIBRARY techmap;
    USE techmap.gencomp.ALL;
LIBRARY grlib;
    USE grlib.amba.ALL;

PACKAGE config IS

    ------------------------------------------------------------------------------
    -- Synthesis settings
    ------------------------------------------------------------------------------
    CONSTANT CFG_TECH                   : integer                       := spartan3e;
    ------------------------------------------------------------------------------
    -- Component settings
    ------------------------------------------------------------------------------
    ------------------------------------------------------------------------------
    -- mctrl
    CONSTANT CFG_MCTRL_HINDEX           : integer                       := 3;
    CONSTANT CFG_MCTRL_PINDEX           : integer                       := 2;
    CONSTANT CFG_MCTRL_ROMADDR          : integer                       := 16#000#;
    CONSTANT CFG_MCTRL_ROMMASK          : integer                       := 16#e00#;
    CONSTANT CFG_MCTRL_IOADDR           : integer                       := 16#200#;
    CONSTANT CFG_MCTRL_IOMASK           : integer                       := 16#e00#;
    CONSTANT CFG_MCTRL_RAMADDR          : integer                       := 16#400#;
    CONSTANT CFG_MCTRL_RAMMASK          : integer                       := 16#c00#;
    CONSTANT CFG_MCTRL_PADDR            : integer                       := 16#002#;
    CONSTANT CFG_MCTRL_PMASK            : integer                       := 16#fff#;
    CONSTANT CFG_MCTRL_WPROT            : integer                       := 0;
    CONSTANT CFG_MCTRL_INVCLK           : integer                       := 1;
    CONSTANT CFG_MCTRL_FAST             : integer                       := 0; --slow address decoding for SDRAM
    CONSTANT CFG_MCTRL_ROMASEL          : integer                       := 28;
    CONSTANT CFG_MCTRL_SDRASEL          : integer                       := 29;
    CONSTANT CFG_MCTRL_SRBANKS          : integer                       := 1;
    CONSTANT CFG_MCTRL_RAM8             : integer                       := 0;
    CONSTANT CFG_MCTRL_RAM16            : integer                       := 1; --16 bit memory bus access
    CONSTANT CFG_MCTRL_SDEN             : integer                       := 1; --enables SDRAM controller
    CONSTANT CFG_MCTRL_SEPBUS           : integer                       := 0; --shared memory busses
    CONSTANT CFG_MCTRL_SDBITS           : integer                       := 16;
    CONSTANT CFG_MCTRL_SDLSB            : integer                       := 1; --additional SDRAM address bits (offset)
    CONSTANT CFG_MCTRL_OEPOL            : integer                       := 0; --active low signals
    CONSTANT CFG_MCTRL_SYNCRST          : integer                       := 0;
    CONSTANT CFG_MCTRL_PAGEBURST        : integer                       := 1;
    -- mctrl regs
    CONSTANT CFG_MCTRL_REG_ROMBWIDTH    : std_logic_vector(1 downto 0)  := "01"; --16 bit prom access
    -- mctrl pad
    CONSTANT CFG_MCTRL_PAD_nbDBitsFlash : integer                       := 16;
    CONSTANT CFG_MCTRL_PAD_nbDBitsSDRAM : integer                       := 16;
    CONSTANT CFG_MCTRL_PAD_nbABitsFlash : integer                       := 24;
    CONSTANT CFG_MCTRL_PAD_nbABitsSDRAM : integer                       := 14; --including BA0 and BA1
    --memory simulations
    CONSTANT CFG_MEMSIM_nbABitsFlash    : integer                       := 16;
    CONSTANT CFG_MEMSIM_nbAROWBitsSDRAM : integer                       := 12;
    CONSTANT CFG_MEMSIM_nbACOLBitsSDRAM : integer                       := 9;
    CONSTANT CFG_MEMSIM_nbABABitsSDRAM  : integer                       := 2;
    CONSTANT CFG_MEMSIM_nbDBitsSDRAM    : integer                       := 16;
    ------------------------------------------------------------------------------
    -- ahbuart
    CONSTANT CFG_AHBUART_HINDEX         : integer                       := 6;
    CONSTANT CFG_AHBUART_PINDEX         : integer                       := 0;
    CONSTANT CFG_AHBUART_PADDR          : integer                       := 16#000#;
    CONSTANT CFG_AHBUART_PMASK          : integer                       := 16#fff#;
    ------------------------------------------------------------------------------
```

```
  86    -- apbuart
  87    CONSTANT CFG_APBUART_PINDEX         : integer                       := 1;
  88    CONSTANT CFG_APBUART_PADDR          : integer                       := 16#001#;
  89    CONSTANT CFG_APBUART_PMASK          : integer                       := 16#fff#;
  90    CONSTANT CFG_APBUART_CONSOLE        : integer                       := 0;
  91    CONSTANT CFG_APBUART_PIRQ           : integer                       := 1;
  92    CONSTANT CFG_APBUART_PARITY         : integer                       := 1;
  93    CONSTANT CFG_APBUART_FLOW           : integer                       := 1;
  94    CONSTANT CFG_APBUART_FIFOSIZE       : integer range 1 to 32         := 8;
  95    CONSTANT CFG_APBUART_ABITS          : integer                       := 8;
  96    ----------------------------------------------------------------------------
  97    -- dsu3
  98    CONSTANT CFG_DSU3_HINDEX            : integer                       := 2;
  99    CONSTANT CFG_DSU3_HADDR             : integer                       := 16#801#;
 100    CONSTANT CFG_DSU3_HMASK             : integer                       := 16#fff#;
 101    CONSTANT CFG_DSU3_NCPU              : integer                       := 1;
 102    CONSTANT CFG_DSU3_TBITS             : integer                       := 30;
 103    CONSTANT CFG_DSU3_TECH              : integer                       := CFG_TECH;
 104    CONSTANT CFG_DSU3_IRQ               : integer                       := 2;
 105    CONSTANT CFG_DSU3_KBYTES            : integer                       := 4;
 106    ----------------------------------------------------------------------------
 107    -- greth
 108    CONSTANT CFG_GRETH_HINDEX           : integer                       := 4;
 109    CONSTANT CFG_GRETH_PINDEX           : integer                       := 3;
 110    CONSTANT CFG_GRETH_PADDR            : integer                       := 16#003#;
 111    CONSTANT CFG_GRETH_PMASK            : integer                       := 16#fff#;
 112    CONSTANT CFG_GRETH_PIRQ             : integer                       := 3;
 113    CONSTANT CFG_GRETH_MEMTECH          : integer                       := CFG_TECH;
 114    CONSTANT CFG_GRETH_IFG_GAP          : integer                       := 24;
 115    CONSTANT CFG_GRETH_ATTEMPT_LIMIT    : integer                       := 16;
 116    CONSTANT CFG_GRETH_BACKOFF_LIMIT    : integer                       := 10;
 117    CONSTANT CFG_GRETH_SLOT_TIME        : integer                       := 128;
 118    CONSTANT CFG_GRETH_MDCSCALER        : integer range 0 to 255        := 25;
 119    CONSTANT CFG_GRETH_ENABLE_MDIO      : integer range 0 to 1          := 0;
 120    CONSTANT CFG_GRETH_FIFOSIZE         : integer range 4 to 64         := 4;
 121    CONSTANT CFG_GRETH_NSYNC            : integer range 1 to 2          := 2;
 122    CONSTANT CFG_GRETH_EDCL             : integer range 0 to 1          := 0;
 123    CONSTANT CFG_GRETH_EDCLBUFSZ        : integer range 1 to 64         := 1;
 124    CONSTANT CFG_GRETH_MACADDRH         : integer                       := 16#00005e#;
 125    CONSTANT CFG_GRETH_MACADDRL         : integer                       := 16#000000#;
 126    CONSTANT CFG_GRETH_IPADDRH          : integer                       := 16#c0a8#;
 127    CONSTANT CFG_GRETH_IPADDRL          : integer                       := 16#0035#;
 128    CONSTANT CFG_GRETH_PHYRSTADR        : integer range 0 to 31         := 0;
 129    CONSTANT CFG_GRETH_RMII             : integer range 0 to 1          := 0;
 130    CONSTANT CFG_GRETH_OEPOL            : integer range 0 to 1          := 0;
 131    CONSTANT CFG_GRETH_SCANEN           : integer range 0 to 1          := 0;
 132    -- greth pad
 133    CONSTANT CFG_ETH_BUS_WIDTH          : integer                       := 8;
 134    CONSTANT USE_ETH                    : integer                       := 0; --0=not used
 135        ----------------------------------------------------------------------------
 136    -- grgpio
 137    CONSTANT CFG_GRGPIO_PINDEX          : integer                       := 5;
 138    CONSTANT CFG_GRGPIO_PADDR           : integer                       := 16#005#;
 139    CONSTANT CFG_GRGPIO_PMASK           : integer                       := 16#fff#;
 140    CONSTANT CFG_GRGPIO_IMASK           : integer                       := 16#0000#;
 141    CONSTANT CFG_GRGPIO_NBITS           : integer                       := 17; --number of I/O signals
 142    CONSTANT CFG_GRGPIO_OEPOL           : integer                       := 0;
 143    CONSTANT CFG_GRGPIO_SYNCRST         : integer                       := 0;
 144    CONSTANT CFG_GRGPIO_BYPASS          : integer                       := 16#0000#;
 145    --grgpio pad
 146    CONSTANT CFG_GRGPIO_BUS_WIDTH       : integer                       := 17;
 147    ----------------------------------------------------------------------------
 148    -- irqmp
 149    CONSTANT CFG_IRQMP_PINDEX           : integer                       := 6;
 150    CONSTANT CFG_IRQMP_PADDR            : integer                       := 16#006#;
 151    CONSTANT CFG_IRQMP_PMASK            : integer                       := 16#fff#;
 152    CONSTANT CFG_IRQMP_NCPU             : integer                       := 1;
 153    CONSTANT CFG_IRQMP_CMASK            : integer                       := 16#fff#;
 154    ----------------------------------------------------------------------------
 155    -- leon3s
 156    CONSTANT CFG_LEON3S_HINDEX          : integer                       := 1;
 157    CONSTANT CFG_LEON3S_FABTECH         : integer range 0 to ntech      := CFG_TECH;
 158    CONSTANT CFG_LEON3S_MEMTECH         : integer range 0 to ntech      := CFG_TECH;
 159    CONSTANT CFG_LEON3S_NWINDOWS        : integer range 2 to 32         := 8; --nb of registers
 160    CONSTANT CFG_LEON3S_DSU             : integer range 0 to 1          := 1; --activate DSU module
 161    CONSTANT CFG_LEON3S_FPU             : integer range 0 to 31         := 0;
 162    CONSTANT CFG_LEON3S_V8              : integer range 0 to 2          := 0;
 163    CONSTANT CFG_LEON3S_CP              : integer range 0 to 1          := 0;
 164    CONSTANT CFG_LEON3S_MAC             : integer range 0 to 1          := 0;
 165    CONSTANT CFG_LEON3S_PCLOW           : integer range 0 to 2          := 2;
 166    CONSTANT CFG_LEON3S_NOTAG           : integer range 0 to 1          := 0;
 167    CONSTANT CFG_LEON3S_NWP             : integer range 0 to 4          := 4; --nb of watchpoints
 168    CONSTANT CFG_LEON3S_ICEN            : integer range 0 to 1          := 0; --disable instr. cache
 169    CONSTANT CFG_LEON3S_IREPL           : integer range 0 to 2          := 0;
 170    CONSTANT CFG_LEON3S_ISETS           : integer range 1 to 4          := 1;
```

```
171     CONSTANT CFG_LEON3S_ILINESIZE       : integer range 4 to 8          := 4;
172     CONSTANT CFG_LEON3S_ISETSIZE        : integer range 1 to 256        := 4;
173     CONSTANT CFG_LEON3S_ISETLOCK        : integer range 0 to 1          := 0;
174     CONSTANT CFG_LEON3S_DCEN            : integer range 0 to 1          := 0; --disable data cache
175     CONSTANT CFG_LEON3S_DREPL           : integer range 0 to 2          := 0;
176     CONSTANT CFG_LEON3S_DSETS           : integer range 1 to 4          := 1;
177     CONSTANT CFG_LEON3S_DLINESIZE       : integer range 4 to 8          := 4;
178     CONSTANT CFG_LEON3S_DSETSIZE        : integer range 1 to 256        := 4;
179     CONSTANT CFG_LEON3S_DSETLOCK        : integer range 0 to 1          := 0;
180     CONSTANT CFG_LEON3S_DSNOOP          : integer range 0 to 6          := 2;
181     CONSTANT CFG_LEON3S_ILRAM           : integer range 0 to 1          := 0;
182     CONSTANT CFG_LEON3S_ILRAMSIZE       : integer range 1 to 512        := 1;
183     CONSTANT CFG_LEON3S_ILRAMSTART      : integer range 0 to 255        := 16#8e#;
184     CONSTANT CFG_LEON3S_DLRAM           : integer range 0 to 1          := 0;
185     CONSTANT CFG_LEON3S_DLRAMSIZE       : integer range 1 to 512        := 1;
186     CONSTANT CFG_LEON3S_DLRAMSTART      : integer range 0 to 255        := 16#8f#;
187     CONSTANT CFG_LEON3S_MMUEN           : integer range 0 to 1          := 0;
188     CONSTANT CFG_LEON3S_ITLBNUM         : integer range 2 to 64         := 8;
189     CONSTANT CFG_LEON3S_DTLBNUM         : integer range 2 to 64         := 8;
190     CONSTANT CFG_LEON3S_TLB_TYPE        : integer range 0 to 1          := 1;
191     CONSTANT CFG_LEON3S_TLB_REP         : integer range 0 to 1          := 0;
192     CONSTANT CFG_LEON3S_LDDEL           : integer range 1 to 2          := 2;
193     CONSTANT CFG_LEON3S_DISAS           : integer range 0 to 2          := 0;
194     CONSTANT CFG_LEON3S_TBUF            : integer range 0 to 64         := 0; --8
195     CONSTANT CFG_LEON3S_PWD             : integer range 0 to 2          := 0;
196     CONSTANT CFG_LEON3S_SVT             : integer range 0 to 1          := 0;
197     CONSTANT CFG_LEON3S_RSTADDR         : integer                       := 0;
198     CONSTANT CFG_LEON3S_SMP             : integer range 0 to 15         := 0;
199     CONSTANT CFG_LEON3S_CACHED          : integer                       := 0;
200     CONSTANT CFG_LEON3S_SCANTEST        : integer                       := 0;
201     ------------------------------------------------------------------------------
202     -- ahbctrl
203     CONSTANT CFG_AHBCTRL_DEFMAST        : integer                       := 0;
204     CONSTANT CFG_AHBCTRL_SPLIT          : integer                       := 0;
205     CONSTANT CFG_AHBCTRL_RROBIN         : integer                       := 1; --round robin mode
206     CONSTANT CFG_AHBCTRL_TIMEOUT        : integer range 0 to 255        := 0;
207     CONSTANT CFG_AHBCTRL_IOADDR         : ahb_addr_type                 := 16#fff#;
208     CONSTANT CFG_AHBCTRL_IOMASK         : ahb_addr_type                 := 16#fff#;
209     CONSTANT CFG_AHBCTRL_CFGADDR        : ahb_addr_type                 := 16#ff0#;
210     CONSTANT CFG_AHBCTRL_CFGMASK        : ahb_addr_type                 := 16#ff0#;
211     CONSTANT CFG_AHBCTRL_NAHBM          : integer range 1 to nahbmst    := nahbmst;
212     CONSTANT CFG_AHBCTRL_NAHBS          : integer range 1 to nahbslv    := nahbslv;
213     CONSTANT CFG_AHBCTRL_IOEN           : integer range 0 to 15         := 1;
214     CONSTANT CFG_AHBCTRL_DISIRQ         : integer range 0 to 1          := 0;
215     CONSTANT CFG_AHBCTRL_FIXBRST        : integer range 0 to 1          := 0;
216     CONSTANT CFG_AHBCTRL_DEBUG          : integer range 0 to 2          := 2;
217     CONSTANT CFG_AHBCTRL_FPNPEN         : integer range 0 to 1          := 1;
218     CONSTANT CFG_AHBCTRL_ICHECK         : integer range 0 to 1          := 1;
219     CONSTANT CFG_AHBCTRL_DEVID          : integer                       := 0;
220     CONSTANT CFG_AHBCTRL_ENBUSMON       : integer range 0 to 1          := 0;
221     CONSTANT CFG_AHBCTRL_ASSERTWARN     : integer range 0 to 1          := 0;
222     CONSTANT CFG_AHBCTRL_ASSERTERR      : integer range 0 to 1          := 0;
223     CONSTANT CFG_AHBCTRL_HMSTDISABLE    : integer                       := 0;
224     CONSTANT CFG_AHBCTRL_HSLVDISABLE    : integer                       := 0;
225     CONSTANT CFG_AHBCTRL_ARBDISABLE     : integer                       := 0;
226     ------------------------------------------------------------------------------
227     -- apbctrl
228     CONSTANT CFG_APBCTRL_HINDEX         : integer                       := 7;
229     CONSTANT CFG_APBCTRL_HADDR          : integer                       := 16#800#;
230     CONSTANT CFG_APBCTRL_HMASK          : integer                       := 16#fff#;
231     CONSTANT CFG_APBCTRL_NSLAVES        : integer range 1 to napbslv    := napbslv;
232     CONSTANT CFG_APBCTRL_DEBUG          : integer range 0 to 2          := 2;
233     CONSTANT CFG_APBCTRL_ICHECK         : integer range 0 to 1          := 1;
234     CONSTANT CFG_APBCTRL_ENBUSMON       : integer range 0 to 1          := 0;
235     CONSTANT CFG_APBCTRL_ASSERTERR      : integer range 0 to 1          := 0;
236     CONSTANT CFG_APBCTRL_ASSERTWARN     : integer range 0 to 1          := 0;
237     CONSTANT CFG_APBCTRL_PSLVDISABLE    : integer                       := 0;
238
239 END;
240
```

Beilage 14

toplevel_arch_struct.vhd - IP Cores & I/Os

```
 1 --
 2 -- VHDL Architecture ambarchitect.toplevel_arch.struct
 3 --
 4 -- Created:
 5 --          by - Thomas.UNKNOWN (WE2778)
 6 --          at - 15:35:00 30.06.2009
 7 --
 8 -- Generated by Mentor Graphics' HDL Designer(TM) 2007.1a (Build 13)
 9 --
10 library ieee;
11 use ieee.std_logic_1164.all;
12 use ieee.numeric_std.all;
13 library grlib;
14 use grlib.stdlib.all;
15 use grlib.amba.all;
16 use grlib.devices.all;
17 library std;
18 use std.textio.all;
19 library techmap;
20 use techmap.gencomp.all;
21 library gaisler;
22 use gaisler.misc.all;
23 use gaisler.uart.all;
24 use gaisler.libdcom.all;
25 use gaisler.memctrl.all;
26 use gaisler.arith.all;
27 use gaisler.leon3.all;
28 use gaisler.libiu.all;
29 use gaisler.net.all;
30 use gaisler.ethernet_mac.all;
31 use gaisler.libcache.all;
32 use gaisler.libproc3.all;
33 library esa;
34 use esa.memoryctrl.all;
35 use techmap.allmem.all;
36 library ambarchitect;
37 use ambarchitect.config.all;
38 USE techmap.allpads.all;
39
40 LIBRARY esa;
41 LIBRARY gaisler;
42 LIBRARY grlib;
43
44 ARCHITECTURE struct OF toplevel_arch IS
45
46    -- Architecture declarations
47
48    -- Internal signal declarations
49    SIGNAL ahbmi   : ahb_mst_in_type;
50    SIGNAL ahbmo   : ahb_mst_out_vector := (OTHERS => ahbm_none);
51    SIGNAL ahbsi   : ahb_slv_in_type;
52    SIGNAL ahbso   : ahb_slv_out_vector := (OTHERS => ahbs_none);
53    SIGNAL apbi    : apb_slv_in_type;
54    SIGNAL apbo    : apb_slv_out_vector := (OTHERS => apb_none);
55    SIGNAL dbgi    : l3_debug_out_vector(0 TO CFG_DSU3_NCPU-1);
56    SIGNAL dbgo    : l3_debug_in_vector(0 TO CFG_DSU3_NCPU-1);
57    SIGNAL dsui    : dsu_in_type;
58    SIGNAL dsuo    : dsu_out_type;
59    SIGNAL ethi    : eth_in_type;
60    SIGNAL etho    : eth_out_type;
```

```
   61     SIGNAL gpioi   : gpio_in_type;
   62     SIGNAL gpioo   : gpio_out_type;
   63     SIGNAL irqi    : irq_out_vector(0 TO CFG_IRQMP_NCPU-1);
   64     SIGNAL irqo    : irq_in_vector(0 TO CFG_IRQMP_NCPU-1);
   65     SIGNAL memi    : memory_in_type;
   66     SIGNAL memo    : memory_out_type;
   67     SIGNAL scanen  : std_ulogic         := '0';
   68     SIGNAL sdo     : sdram_out_type;
   69     SIGNAL testen  : std_ulogic         := '0';
   70     SIGNAL testoen : std_ulogic         := '1';
   71     SIGNAL testrst : std_ulogic         := '1';
   72     SIGNAL uarti   : uart_in_type;
   73     SIGNAL uarti1  : uart_in_type;
   74     SIGNAL uarto   : uart_out_type;
   75     SIGNAL uarto1  : uart_out_type;
   76     SIGNAL wpo     : wprot_out_type;
   77
   78
   79     -- Component Declarations
   80     COMPONENT mctrl
   81     GENERIC (
   82        hindex    : integer;
   83        pindex    : integer;
   84        romaddr   : integer;
   85        rommask   : integer;
   86        ioaddr    : integer;
   87        iomask    : integer;
   88        ramaddr   : integer;
   89        rammask   : integer;
   90        paddr     : integer;
   91        pmask     : integer;
   92        wprot     : integer;
   93        invclk    : integer;
   94        fast      : integer;
   95        romasel   : integer;
   96        sdrasel   : integer;
   97        srbanks   : integer;
   98        ram8      : integer;
   99        ram16     : integer;
  100        sden      : integer;
  101        sepbus    : integer;
  102        sdbits    : integer;
  103        sdlsb     : integer;      -- set to 12 for the GE-HPE board
  104        oepol     : integer;
  105        syncrst   : integer;
  106        pageburst : integer;
  107        scantest  : integer;
  108        mobile    : integer
  109     );
  110     PORT (
  111        ahbsi : IN     ahb_slv_in_type;
  112        apbi  : IN     apb_slv_in_type;
  113        clk   : IN     std_ulogic;
  114        memi  : IN     memory_in_type;
  115        rst   : IN     std_ulogic;
  116        wpo   : IN     wprot_out_type;
  117        ahbso : OUT    ahb_slv_out_type;
  118        apbo  : OUT    apb_slv_out_type;
  119        memo  : OUT    memory_out_type;
  120        sdo   : OUT    sdram_out_type
```

```
121     );
122     END COMPONENT;
123     COMPONENT ahbuart
124     GENERIC (
125        hindex : integer;
126        pindex : integer;
127        paddr  : integer;
128        pmask  : integer
129     );
130     PORT (
131        ahbi  : IN     ahb_mst_in_type;
132        apbi  : IN     apb_slv_in_type;
133        clk   : IN     std_ulogic;
134        rst   : IN     std_ulogic;
135        uarti : IN     uart_in_type;
136        ahbo  : OUT    ahb_mst_out_type;
137        apbo  : OUT    apb_slv_out_type;
138        uarto : OUT    uart_out_type
139     );
140     END COMPONENT;
141     COMPONENT apbuart
142     GENERIC (
143        pindex   : integer;
144        paddr    : integer;
145        pmask    : integer;
146        console  : integer;
147        pirq     : integer;
148        parity   : integer;
149        flow     : integer;
150        fifosize : integer range 1 to 32;
151        abits    : integer
152     );
153     PORT (
154        apbi  : IN     apb_slv_in_type;
155        clk   : IN     std_ulogic;
156        rst   : IN     std_ulogic;
157        uarti : IN     uart_in_type;
158        apbo  : OUT    apb_slv_out_type;
159        uarto : OUT    uart_out_type
160     );
161     END COMPONENT;
162     COMPONENT dsu3
163     GENERIC (
164        hindex : integer;
165        haddr  : integer;
166        hmask  : integer;
167        ncpu   : integer;
168        tbits  : integer;      -- timer bits (instruction trace time
    tag)
169        tech   : integer;
170        irq    : integer;
171        kbytes : integer
172     );
173     PORT (
174        ahbmi : IN     ahb_mst_in_type;
175        ahbsi : IN     ahb_slv_in_type;
176        clk   : IN     std_ulogic;
177        dbgi  : IN     l3_debug_out_vector (0 TO NCPU-1);
178        dsui  : IN     dsu_in_type;
179        rst   : IN     std_ulogic;
```

```
180        ahbso : OUT    ahb_slv_out_type;
181        dbgo  : OUT    l3_debug_in_vector (0 TO NCPU-1);
182        dsuo  : OUT    dsu_out_type
183     );
184     END COMPONENT;
185     COMPONENT greth
186     GENERIC (
187        hindex        : integer;
188        pindex        : integer;
189        paddr         : integer;
190        pmask         : integer;
191        pirq          : integer;
192        memtech       : integer;
193        ifg_gap       : integer;
194        attempt_limit : integer;
195        backoff_limit : integer;
196        slot_time     : integer;
197        mdcscaler     : integer range 0 to 255;
198        enable_mdio   : integer range 0 to 1;
199        fifosize      : integer range 4 to 512;
200        nsync         : integer range 1 to 2;
201        edcl          : integer range 0 to 2;
202        edclbufsz     : integer range 1 to 64;
203        macaddrh      : integer;
204        macaddrl      : integer;
205        ipaddrh       : integer;
206        ipaddrl       : integer;
207        phyrstadr     : integer range 0 to 32;
208        rmii          : integer range 0 to 1;
209        oepol         : integer range 0 to 1;
210        scanen        : integer range 0 to 1;
211        ft            : integer range 0 to 2;
212        mdint_pol     : integer range 0 to 1;
213        enable_mdint  : integer range 0 to 1
214     );
215     PORT (
216        ahbmi : IN     ahb_mst_in_type;
217        apbi  : IN     apb_slv_in_type;
218        clk   : IN     std_ulogic;
219        ethi  : IN     eth_in_type;
220        rst   : IN     std_ulogic;
221        ahbmo : OUT    ahb_mst_out_type;
222        apbo  : OUT    apb_slv_out_type;
223        etho  : OUT    eth_out_type
224     );
225     END COMPONENT;
226     COMPONENT grgpio
227     GENERIC (
228        pindex   : integer;
229        paddr    : integer;
230        pmask    : integer;
231        imask    : integer;
232        nbits    : integer;      -- GPIO bits
233        oepol    : integer;      -- Output enable polarity
234        syncrst  : integer;      -- Only synchronous reset
235        bypass   : integer;
236        scantest : integer;
237        bpdir    : integer
238     );
239     PORT (
```

```
240         apbi  : IN     apb_slv_in_type;
241         clk   : IN     std_ulogic;
242         gpioi : IN     gpio_in_type;
243         rst   : IN     std_ulogic;
244         apbo  : OUT    apb_slv_out_type;
245         gpioo : OUT    gpio_out_type
246       );
247       END COMPONENT;
248       COMPONENT irqmp
249       GENERIC (
250         pindex : integer;
251         paddr  : integer;
252         pmask  : integer;
253         ncpu   : integer;
254         eirq   : integer
255       );
256       PORT (
257         apbi : IN     apb_slv_in_type;
258         clk  : IN     std_ulogic;
259         irqi : IN     irq_out_vector (0 TO ncpu-1);
260         rst  : IN     std_ulogic;
261         apbo : OUT    apb_slv_out_type;
262         irqo : OUT    irq_in_vector (0 TO ncpu-1)
263       );
264       END COMPONENT;
265       COMPONENT leon3s
266       GENERIC (
267         hindex     : integer;
268         fabtech    : integer range 0 to NTECH;
269         memtech    : integer range 0 to NTECH;
270         nwindows   : integer range 2 to 32;
271         dsu        : integer range 0 to 1;
272         fpu        : integer range 0 to 31;
273         v8         : integer range 0 to 63;
274         cp         : integer range 0 to 1;
275         mac        : integer range 0 to 1;
276         pclow      : integer range 0 to 2;
277         notag      : integer range 0 to 1;
278         nwp        : integer range 0 to 4;
279         icen       : integer range 0 to 1;
280         irepl      : integer range 0 to 2;
281         isets      : integer range 1 to 4;
282         ilinesize  : integer range 4 to 8;
283         isetsize   : integer range 1 to 256;
284         isetlock   : integer range 0 to 1;
285         dcen       : integer range 0 to 1;
286         drepl      : integer range 0 to 2;
287         dsets      : integer range 1 to 4;
288         dlinesize  : integer range 4 to 8;
289         dsetsize   : integer range 1 to 256;
290         dsetlock   : integer range 0 to 1;
291         dsnoop     : integer range 0 to 6;
292         ilram      : integer range 0 to 1;
293         ilramsize  : integer range 1 to 512;
294         ilramstart : integer range 0 to 255;
295         dlram      : integer range 0 to 1;
296         dlramsize  : integer range 1 to 512;
297         dlramstart : integer range 0 to 255;
298         mmuen      : integer range 0 to 1;
299         itlbnum    : integer range 2 to 64;
```

```
300       dtlbnum    : integer range 2 to 64;
301       tlb_type   : integer range 0 to 3;
302       tlb_rep    : integer range 0 to 1;
303       lddel      : integer range 1 to 2;
304       disas      : integer range 0 to 2;
305       tbuf       : integer range 0 to 64;
306       pwd        : integer range 0 to 2;       -- power-down
307       svt        : integer range 0 to 1;       -- single vector
    trapping
308       rstaddr    : integer;
309       smp        : integer range 0 to 15;      -- support SMP systems
310       cached     : integer;                    -- cacheability table
311       scantest   : integer
312    );
313    PORT (
314       ahbi  : IN     ahb_mst_in_type;
315       ahbsi : IN     ahb_slv_in_type;
316       ahbso : IN     ahb_slv_out_vector;
317       clk   : IN     std_ulogic;
318       dbgi  : IN     l3_debug_in_type;
319       irqi  : IN     l3_irq_in_type;
320       rstn  : IN     std_ulogic;
321       ahbo  : OUT    ahb_mst_out_type;
322       dbgo  : OUT    l3_debug_out_type;
323       irqo  : OUT    l3_irq_out_type
324    );
325    END COMPONENT;
326    COMPONENT ahbctrl
327    GENERIC (
328       defmast     : integer;                          -- default
    master
329       split       : integer;                          -- split support
330       rrobin      : integer;                          -- round-robin
    arbitration
331       timeout     : integer range 0 to 255;           -- HREADY
    timeout
332       ioaddr      : ahb_addr_type;                    -- I/O area MSB
    address
333       iomask      : ahb_addr_type;                    -- I/O area
    address mask
334       cfgaddr     : ahb_addr_type;                    -- config area
    MSB address
335       cfgmask     : ahb_addr_type;                    -- config area
    address mask
336       nahbm       : integer range 1 to NAHBMST;       -- number of
    masters
337       nahbs       : integer range 1 to NAHBSLV;       -- number of
    slaves
338       ioen        : integer range 0 to 15;            -- enable I/O
    area
339       disirq      : integer range 0 to 1;             -- disable
    interrupt routing
340       fixbrst     : integer range 0 to 1;             -- support
    fix-length bursts
341       debug       : integer range 0 to 2;             -- report cores
    to console
342       fpnpen      : integer range 0 to 1;             -- full PnP
    configuration decoding
343       icheck      : integer range 0 to 1;
344       devid       : integer;                          -- unique device
```

```
      ID
  345       enbusmon    : integer range 0 to 1;            --enable bus
      monitor
  346       assertwarn  : integer range 0 to 1;            --enable
      assertions for warnings
  347       asserterr   : integer range 0 to 1;            --enable
      assertions for errors
  348       hmstdisable : integer;                         --disable master
      checks
  349       hslvdisable : integer;                         --disable slave
      checks
  350       arbdisable  : integer;                         --disable
      arbiter checks
  351       mprio       : integer;                         --master with
      highest priority
  352       enebterm    : integer range 0 to 1             --enable early
      burst termination
  353     );
  354     PORT (
  355        clk     : IN     std_ulogic;
  356        msto    : IN     ahb_mst_out_vector;
  357        rst     : IN     std_ulogic;
  358        scanen  : IN     std_ulogic  := '0';
  359        slvo    : IN     ahb_slv_out_vector;
  360        testen  : IN     std_ulogic  := '0';
  361        testoen : IN     std_ulogic  := '1';
  362        testrst : IN     std_ulogic  := '1';
  363        msti    : OUT    ahb_mst_in_type;
  364        slvi    : OUT    ahb_slv_in_type
  365     );
  366     END COMPONENT;
  367     COMPONENT apbctrl
  368     GENERIC (
  369        hindex      : integer;
  370        haddr       : integer;
  371        hmask       : integer;
  372        nslaves     : integer range 1 to NAPBSLV;
  373        debug       : integer range 0 to 2;
  374        icheck      : integer range 0 to 1;
  375        enbusmon    : integer range 0 to 1;
  376        asserterr   : integer range 0 to 1;
  377        assertwarn  : integer range 0 to 1;
  378        pslvdisable : integer
  379     );
  380     PORT (
  381        ahbi : IN     ahb_slv_in_type;
  382        apbo : IN     apb_slv_out_vector;
  383        clk  : IN     std_ulogic;
  384        rst  : IN     std_ulogic;
  385        ahbo : OUT    ahb_slv_out_type;
  386        apbi : OUT    apb_slv_in_type
  387     );
  388     END COMPONENT;
  389
  390     -- Optional embedded configurations
  391     -- pragma synthesis_off
  392     FOR ALL : ahbctrl USE ENTITY grlib.ahbctrl;
  393     FOR ALL : ahbuart USE ENTITY gaisler.ahbuart;
  394     FOR ALL : apbctrl USE ENTITY grlib.apbctrl;
  395     FOR ALL : apbuart USE ENTITY gaisler.apbuart;
```

```
396     FOR ALL : dsu3 USE ENTITY gaisler.dsu3;
397     FOR ALL : grgpio USE ENTITY gaisler.grgpio;
398     FOR ALL : irqmp USE ENTITY gaisler.irqmp;
399     FOR ALL : leon3s USE ENTITY gaisler.leon3s;
400     FOR ALL : mctrl USE ENTITY esa.mctrl;
401     -- pragma synthesis_on
402
403
404  BEGIN
405     -- Architecture concurrent statements
406     -- HDL Embedded Text Block 1 eb1
407     scanen <= '0';
408     testen <= '0';
409     testrst <= '1';
410     testoen <= '1';
411
412     -- HDL Embedded Text Block 2 mctrl_signal_routes
413     mctrl_pad_flash_cs_n <= memo.romsn(0);
414     mctrl_pad_flash_oe_n <= memo.oen;
415     mctrl_pad_flash_we_n <= memo.writen;
416     mctrl_pad_flashResetn <= '1';
417     memi.bwidth <= CFG_MCTRL_REG_ROMBWIDTH;
418
419
420     mctrl_pad_sd_cas_n <=  sdo.casn;
421     mctrl_pad_sd_cke <= sdo.sdcke(0);
422     mctrl_pad_sd_dqmh <= sdo.dqm(4);
423     mctrl_pad_sd_dqml <= sdo.dqm(0);
424     mctrl_pad_sd_ras_n <= sdo.rasn;
425
426     mctrl_pad_sd_cs_n <= sdo.sdcsn(0);
427     --mctrl_pad_sd_cs_n <= memo.ramsn(4);
428
429     mctrl_pad_sd_we_n <= sdo.sdwen;
430     --mctrl_pad_sd_we_n <= memo.writen;
431
432
433
434     --mctrl_pad_addr <= memo.address(CFG_MCTRL_PAD_nbABitsFlash downto
      1); --sim
435     mctrl_pad_addr <= memo.address(CFG_MCTRL_PAD_nbABitsFlash-1 downto
      0); --fpga
436
437
438     memiproc: process (clk, rstn)
439     begin
440     if rstn = '0' then
441       memi.data <= (others => 'Z');
442     elsif rising_edge(clk) then
443       if memo.romsn(0) = '0' then
444         memi.data(31 downto 16) <= mctrl_pad_data_in;
445         memi.data(15 downto 0) <= (others => '0');
446       elsif sdo.sdcsn(0) = '0' then
447         memi.data(15 downto 0) <= mctrl_pad_data_in;
448         memi.data(31 downto 16) <= (others => '0');
449       else
450         memi.data <= (others => 'Z');
451       end if;
452     end if;
453     end process memiproc;
```

```
454
455
456     --memi.data(31 downto 16) <= mctrl_pad_data_in;
457     --memi.data(15 downto 0) <= mctrl_pad_data_in;
458     --memi.data(CFG_MCTRL_PAD_nbDBitsFlash-1 downto 0) <=
    mctrl_pad_data_in;
459
460
461     mctrl_pad_data_out <= memo.data(31 downto 16) when sdo.dqm(0) =
    '1' else memo.data(15 downto 0);
462     --mctrl_pad_data_out <= memo.data(31 downto 16) when memo.writen =
    '0' else memo.data(15 downto 0);
463     --mctrl_pad_data_out <= memo.data(CFG_MCTRL_PAD_nbDBitsFlash-1
    downto 0);
464
465     --memi.data(15 downto 0) <= (others => 'Z');    --0
466     --memi.data(31 downto 16) <= (others => 'Z');
467
468
469     memi.sd <= (others => 'Z');
470     memi.wrn <= "1111";
471     memi.writen <= '1';
472     wpo.wprothit <= '1';
473     memi.brdyn <= '1';
474     memi.bexcn <= '1';
475
476     -- HDL Embedded Text Block 4 apbuart_signal_routes
477     uarti.rxd <= apbuart_rx;
478     uarti.ctsn <= '1';
479     uarti.extclk <= '0';
480     apbuart_tx <= uarto.txd;
481
482     -- HDL Embedded Text Block 5 ahbuart_signal_routes
483     uarti1.rxd <= ahbuart_rx;
484     uarti1.ctsn <= '1';
485     uarti1.extclk <= '0';
486     ahbuart_tx <= uarto1.txd;
487
488
489     -- HDL Embedded Text Block 6 grgpio_signal_routes
490     gpioi.din(CFG_GRGPIO_BUS_WIDTH-1 downto 0) <= grgpio_pad_in;
491     gpioi.sig_in <= (others =>'Z');
492     grgpio_pad_out <= gpioo.dout(CFG_GRGPIO_BUS_WIDTH-1 DOWNTO 0);
493     grgpio_pad_oen <= gpioo.oen(CFG_GRGPIO_BUS_WIDTH-1 DOWNTO 0);
494
495     -- HDL Embedded Text Block 7 dsu_signal_routes
496     dsui.enable <= dsu_pad_enable;
497     dsui.break <= dsu_pad_break;
498     dsu_pad_active <= dsuo.active;
499
500     --dsui.enable <= '1';
501     --dsui.break <= '1';
502
503
504     -- Instance port mappings.
505     I31 : mctrl
506        GENERIC MAP (
507           hindex    => CFG_MCTRL_HINDEX,
508           pindex    => CFG_MCTRL_PINDEX,
509           romaddr   => CFG_MCTRL_ROMADDR,
```

```
510           rommask   => CFG_MCTRL_ROMMASK,
511           ioaddr    => CFG_MCTRL_IOADDR,
512           iomask    => CFG_MCTRL_IOMASK,
513           ramaddr   => CFG_MCTRL_RAMADDR,
514           rammask   => CFG_MCTRL_RAMMASK,
515           paddr     => CFG_MCTRL_PADDR,
516           pmask     => CFG_MCTRL_PMASK,
517           wprot     => CFG_MCTRL_WPROT,
518           invclk    => CFG_MCTRL_INVCLK,
519           fast      => CFG_MCTRL_FAST,
520           romasel   => CFG_MCTRL_ROMASEL,
521           sdrasel   => CFG_MCTRL_SDRASEL,
522           srbanks   => CFG_MCTRL_SRBANKS,
523           ram8      => CFG_MCTRL_RAM8,
524           ram16     => CFG_MCTRL_RAM16,
525           sden      => CFG_MCTRL_SDEN,
526           sepbus    => CFG_MCTRL_SEPBUS,
527           sdbits    => CFG_MCTRL_SDBITS,
528           sdlsb     => CFG_MCTRL_SDLSB,         -- set to 12 for the
    GE-HPE board
529           oepol     => CFG_MCTRL_OEPOL,
530           syncrst   => CFG_MCTRL_SYNCRST,
531           pageburst => CFG_MCTRL_PAGEBURST,
532           scantest  => 0,
533           mobile    => 0
534        )
535        PORT MAP (
536           rst => rstn,
537           clk => clk,
538           memi => memi,
539           memo => memo,
540           ahbsi => ahbsi,
541           apbi => apbi,
542           wpo => wpo,
543           sdo => sdo,
544           ahbso => ahbso(CFG_MCTRL_HINDEX),
545           apbo => apbo(CFG_MCTRL_PINDEX)
546        );
547     I6 : ahbuart
548        GENERIC MAP (
549           hindex => CFG_AHBUART_HINDEX,
550           pindex => CFG_AHBUART_PINDEX,
551           paddr  => CFG_AHBUART_PADDR,
552           pmask  => CFG_AHBUART_PMASK
553        )
554        PORT MAP (
555           rst => rstn,
556           clk => clk,
557           uarti => uarti1,
558           uarto => uarto1,
559           apbi => apbi,
560           ahbi => ahbmi,
561           apbo => apbo(CFG_AHBUART_PINDEX),
562           ahbo => ahbmo(CFG_AHBUART_HINDEX)
563        );
564     I8 : apbuart
565        GENERIC MAP (
566           pindex   => CFG_APBUART_PINDEX,
567           paddr    => CFG_APBUART_PADDR,
568           pmask    => CFG_APBUART_PMASK,
```

```
569           console  => CFG_APBUART_CONSOLE,
570           pirq     => CFG_APBUART_PIRQ,
571           parity   => CFG_APBUART_PARITY,
572           flow     => CFG_APBUART_FLOW,
573           fifosize => CFG_APBUART_FIFOSIZE,
574           abits    => CFG_APBUART_ABITS
575        )
576        PORT MAP (
577           rst => rstn,
578           clk => clk,
579           apbi => apbi,
580           uarti => uarti,
581           uarto => uarto,
582           apbo => apbo(CFG_APBUART_PINDEX)
583        );
584     I14 : dsu3
585        GENERIC MAP (
586           hindex => CFG_DSU3_HINDEX,
587           haddr  => CFG_DSU3_HADDR,
588           hmask  => CFG_DSU3_HMASK,
589           ncpu   => CFG_DSU3_NCPU,
590           tbits  => CFG_DSU3_TBITS,         -- timer bits (instruction
    trace time tag)
591           tech   => CFG_DSU3_TECH,
592           irq    => CFG_DSU3_IRQ,
593           kbytes => CFG_DSU3_KBYTES
594        )
595        PORT MAP (
596           rst => rstn,
597           clk => clk,
598           ahbmi => ahbmi,
599           ahbsi => ahbsi,
600           dbgi => dbgi,
601           dbgo => dbgo,
602           dsui => dsui,
603           dsuo => dsuo,
604           ahbso => ahbso(CFG_DSU3_HINDEX)
605        );
606     I18 : grgpio
607        GENERIC MAP (
608           pindex   => CFG_GRGPIO_PINDEX,
609           paddr    => CFG_GRGPIO_PADDR,
610           pmask    => CFG_GRGPIO_PMASK,
611           imask    => CFG_GRGPIO_IMASK,
612           nbits    => CFG_GRGPIO_NBITS,          -- GPIO bits
613           oepol    => CFG_GRGPIO_OEPOL,          -- Output enable
    polarity
614           syncrst  => CFG_GRGPIO_SYNCRST,        -- Only synchronous
    reset
615           bypass   => CFG_GRGPIO_BYPASS,
616           scantest => 0,
617           bpdir    => 16#0000#
618        )
619        PORT MAP (
620           rst => rstn,
621           clk => clk,
622           apbi => apbi,
623           gpioi => gpioi,
624           gpioo => gpioo,
625           apbo => apbo(CFG_GRGPIO_PINDEX)
```

```
626       );
627    I19 : irqmp
628       GENERIC MAP (
629          pindex => CFG_IRQMP_PINDEX,
630          paddr  => CFG_IRQMP_PADDR,
631          pmask  => CFG_IRQMP_PMASK,
632          ncpu   => CFG_IRQMP_NCPU,
633          eirq   => 0
634       )
635       PORT MAP (
636          rst => rstn,
637          clk => clk,
638          apbi => apbi,
639          irqi => irqi,
640          irqo => irqo,
641          apbo => apbo(CFG_IRQMP_PINDEX)
642       );
643    I20 : leon3s
644       GENERIC MAP (
645          hindex     => CFG_LEON3S_HINDEX,
646          fabtech    => CFG_LEON3S_FABTECH,
647          memtech    => CFG_LEON3S_MEMTECH,
648          nwindows   => CFG_LEON3S_NWINDOWS,
649          dsu        => CFG_LEON3S_DSU,
650          fpu        => CFG_LEON3S_FPU,
651          v8         => CFG_LEON3S_V8,
652          cp         => CFG_LEON3S_CP,
653          mac        => CFG_LEON3S_MAC,
654          pclow      => CFG_LEON3S_PCLOW,
655          notag      => CFG_LEON3S_NOTAG,
656          nwp        => CFG_LEON3S_NWP,
657          icen       => CFG_LEON3S_ICEN,
658          irepl      => CFG_LEON3S_IREPL,
659          isets      => CFG_LEON3S_ISETS,
660          ilinesize  => CFG_LEON3S_ILINESIZE,
661          isetsize   => CFG_LEON3S_ISETSIZE,
662          isetlock   => CFG_LEON3S_ISETLOCK,
663          dcen       => CFG_LEON3S_DCEN,
664          drepl      => CFG_LEON3S_DREPL,
665          dsets      => CFG_LEON3S_DSETS,
666          dlinesize  => CFG_LEON3S_DLINESIZE,
667          dsetsize   => CFG_LEON3S_DSETSIZE,
668          dsetlock   => CFG_LEON3S_DSETLOCK,
669          dsnoop     => CFG_LEON3S_DSNOOP,
670          ilram      => CFG_LEON3S_ILRAM,
671          ilramsize  => CFG_LEON3S_ILRAMSIZE,
672          ilramstart => CFG_LEON3S_ILRAMSTART,
673          dlram      => CFG_LEON3S_DLRAM,
674          dlramsize  => CFG_LEON3S_DLRAMSIZE,
675          dlramstart => CFG_LEON3S_DLRAMSTART,
676          mmuen      => CFG_LEON3S_MMUEN,
677          itlbnum    => CFG_LEON3S_ITLBNUM,
678          dtlbnum    => CFG_LEON3S_DTLBNUM,
679          tlb_type   => CFG_LEON3S_TLB_TYPE,
680          tlb_rep    => CFG_LEON3S_TLB_REP,
681          lddel      => CFG_LEON3S_LDDEL,
682          disas      => CFG_LEON3S_DISAS,
683          tbuf       => CFG_LEON3S_TBUF,
684          pwd        => CFG_LEON3S_PWD,           -- power-down
685          svt        => CFG_LEON3S_SVT,           -- single vector
```

```
    trapping
686           rstaddr    => CFG_LEON3S_RSTADDR,
687           smp        => CFG_LEON3S_SMP,            -- support SMP
    systems
688           cached     => CFG_LEON3S_CACHED,         -- cacheability
    table
689           scantest   => CFG_LEON3S_SCANTEST
690         )
691         PORT MAP (
692            clk => clk,
693            rstn => rstn,
694            ahbi => ahbmi,
695            ahbsi => ahbsi,
696            ahbso => ahbso,
697            irqi => irqo(0),
698            irqo => irqi(0),
699            dbgi => dbgo(0),
700            dbgo => dbgi(0),
701            ahbo => ahbmo(CFG_LEON3S_HINDEX)
702         );
703      I0 : ahbctrl
704         GENERIC MAP (
705            defmast     => CFG_AHBCTRL_DEFMAST,            -- default
    master
706            split       => CFG_AHBCTRL_SPLIT,              -- split
    support
707            rrobin      => CFG_AHBCTRL_RROBIN,             --
    round-robin arbitration
708            timeout     => CFG_AHBCTRL_TIMEOUT,            -- HREADY
    timeout
709            ioaddr      => CFG_AHBCTRL_IOADDR,             -- I/O area
    MSB address
710            iomask      => CFG_AHBCTRL_IOMASK,             -- I/O area
    address mask
711            cfgaddr     => CFG_AHBCTRL_CFGADDR,            -- config
    area MSB address
712            cfgmask     => CFG_AHBCTRL_CFGMASK,            -- config
    area address mask
713            nahbm       => CFG_AHBCTRL_NAHBM,              -- number of
    masters
714            nahbs       => CFG_AHBCTRL_NAHBS,              -- number of
    slaves
715            ioen        => CFG_AHBCTRL_IOEN,               -- enable
    I/O area
716            disirq      => CFG_AHBCTRL_DISIRQ,             -- disable
    interrupt routing
717            fixbrst     => CFG_AHBCTRL_FIXBRST,            -- support
    fix-length bursts
718            debug       => CFG_AHBCTRL_DEBUG,              -- report
    cores to console
719            fpnpen      => CFG_AHBCTRL_FPNPEN,             -- full PnP
    configuration decoding
720            icheck      => CFG_AHBCTRL_ICHECK,
721            devid       => CFG_AHBCTRL_DEVID,              -- unique
    device ID
722            enbusmon    => CFG_AHBCTRL_ENBUSMON,           --enable bus
    monitor
723            assertwarn  => CFG_AHBCTRL_ASSERTWARN,         --enable
    assertions for warnings
724            asserterr   => CFG_AHBCTRL_ASSERTERR,          --enable
```

```
      assertions for errors
  725          hmstdisable => CFG_AHBCTRL_HMSTDISABLE,         --disable
      master checks
  726          hslvdisable => CFG_AHBCTRL_HSLVDISABLE,         --disable
      slave checks
  727          arbdisable  => CFG_AHBCTRL_ARBDISABLE,          --disable
      arbiter checks
  728          mprio       => 0,                               --master
      with highest priority
  729          enebterm    => 0                                --enable
      early burst termination
  730       )
  731       PORT MAP (
  732          rst => rstn,
  733          clk => clk,
  734          msti => ahbmi,
  735          msto => ahbmo,
  736          slvi => ahbsi,
  737          slvo => ahbso,
  738          testen => testen,
  739          testrst => testrst,
  740          scanen => scanen,
  741          testoen => testoen
  742       );
  743    I1 : apbctrl
  744       GENERIC MAP (
  745          hindex      => CFG_APBCTRL_HINDEX,
  746          haddr       => CFG_APBCTRL_HADDR,
  747          hmask       => CFG_APBCTRL_HMASK,
  748          nslaves     => CFG_APBCTRL_NSLAVES,
  749          debug       => CFG_APBCTRL_DEBUG,
  750          icheck      => CFG_APBCTRL_ICHECK,
  751          enbusmon    => CFG_APBCTRL_ENBUSMON,
  752          asserterr   => CFG_APBCTRL_ASSERTERR,
  753          assertwarn  => CFG_APBCTRL_ASSERTWARN,
  754          pslvdisable => CFG_APBCTRL_PSLVDISABLE
  755       )
  756       PORT MAP (
  757          rst => rstn,
  758          clk => clk,
  759          ahbi => ahbsi,
  760          apbi => apbi,
  761          apbo => apbo,
  762          ahbo => ahbso(CFG_APBCTRL_HINDEX)
  763       );
  764
  765    g0: IF USE_ETH = 1 GENERATE
  766    -- Optional embedded configurations
  767    -- pragma synthesis_off
  768    FOR ALL : greth USE ENTITY gaisler.greth;
  769    -- pragma synthesis_on
  770
  771    BEGIN
  772       I16 : greth
  773          GENERIC MAP (
  774             hindex        => CFG_GRETH_HINDEX,
  775             pindex        => CFG_GRETH_PINDEX,
  776             paddr         => CFG_GRETH_PADDR,
  777             pmask         => CFG_GRETH_PMASK,
  778             pirq          => CFG_GRETH_PIRQ,
```

```
779             memtech       => CFG_GRETH_MEMTECH,
780             ifg_gap       => CFG_GRETH_IFG_GAP,
781             attempt_limit => CFG_GRETH_ATTEMPT_LIMIT,
782             backoff_limit => CFG_GRETH_BACKOFF_LIMIT,
783             slot_time     => CFG_GRETH_SLOT_TIME,
784             mdcscaler     => CFG_GRETH_MDCSCALER,
785             enable_mdio   => CFG_GRETH_ENABLE_MDIO,
786             fifosize      => CFG_GRETH_FIFOSIZE,
787             nsync         => CFG_GRETH_NSYNC,
788             edcl          => CFG_GRETH_EDCL,
789             edclbufsz     => CFG_GRETH_EDCLBUFSZ,
790             macaddrh      => CFG_GRETH_MACADDRH,
791             macaddrl      => CFG_GRETH_MACADDRL,
792             ipaddrh       => CFG_GRETH_IPADDRH,
793             ipaddrl       => CFG_GRETH_IPADDRL,
794             phyrstadr     => CFG_GRETH_PHYRSTADR,
795             rmii          => CFG_GRETH_RMII,
796             oepol         => CFG_GRETH_OEPOL,
797             scanen        => CFG_GRETH_SCANEN,
798             ft            => 0,
799             mdint_pol     => 0,
800             enable_mdint  => 0
801          )
802          PORT MAP (
803             rst => rstn,
804             clk => clk,
805             ahbmi => ahbmi,
806             apbi => apbi,
807             ethi => ethi,
808             etho => etho,
809             ahbmo => ahbmo(CFG_GRETH_HINDEX),
810             apbo => apbo(CFG_GRETH_PINDEX)
811          );
812       -- HDL Embedded Text Block 3 greth_signal_routes
813       ethi.rx_col <= eth_pad_col;
814       ethi.rx_crs <= eth_pad_crs;
815       ethi.rx_clk <= eth_pad_rx_clk;
816       ethi.rx_dv <= eth_pad_rx_dv;
817       ethi.rx_er <= eth_pad_rx_er;
818       ethi.rxd <= eth_pad_rxd;
819       ethi.tx_clk <= eth_pad_tx_clk;
820       ethi.gtx_clk <= '1';
821       ethi.rmii_clk <= '1';
822       ethi.phyrstaddr <= "00000";
823       ethi.edcladdr <= "0000";
824       eth_pad_mdc <= etho.mdc;
825       eth_pad_reset <= etho.reset;
826       eth_pad_tx_en <= etho.tx_en;
827       eth_pad_txd <= etho.txd;
828       ethi.mdio_i <= eth_pad_mdio_i;
829       eth_pad_mdio_o <= etho.mdio_o;
830
831    END GENERATE g0;
832
833    g1: IF USE_ETH = 0  GENERATE
834    BEGIN
835       -- HDL Embedded Text Block 8 greth_signal_routes1
836
837       eth_pad_mdc <= '-';
838       eth_pad_reset <= '-';
```

```
839         eth_pad_tx_en <= '-';
840         eth_pad_txd <= (others => '-');
841         ethi.mdio_i <= '-';
842         eth_pad_mdio_o <= '-';
843 
844      END GENERATE g1;
845 
846  END struct;
```

Beilage 15

VHDL Code der Pad Blockschemas

```
 1 --
 2 -- VHDL Architecture leon_toplevel.mctrl_pad.struct
 3 --
 4 -- Created:
 5 --          by - Thomas.UNKNOWN (WE2778)
 6 --          at - 16:57:47 29.06.2009
 7 --
 8 -- Generated by Mentor Graphics' HDL Designer(TM) 2007.1a (Build 13)
 9 --
10 LIBRARY ieee;
11 USE ieee.std_logic_1164.all;
12 USE ieee.numeric_std.all;
13 LIBRARY grlib;
14 USE grlib.stdlib.all;
15 USE grlib.amba.all;
16 USE grlib.devices.all;
17 LIBRARY std;
18 USE std.textio.all;
19 LIBRARY techmap;
20 USE techmap.gencomp.all;
21 LIBRARY gaisler;
22 USE gaisler.misc.all;
23 USE gaisler.uart.all;
24 USE gaisler.libdcom.all;
25 USE gaisler.memctrl.all;
26 USE gaisler.arith.all;
27 USE gaisler.leon3.all;
28 USE gaisler.libiu.all;
29 USE gaisler.net.all;
30 USE gaisler.ethernet_mac.all;
31 USE gaisler.libcache.all;
32 USE gaisler.libproc3.all;
33 USE grlib.multlib.all;
34 LIBRARY esa;
35 USE esa.memoryctrl.all;
36 USE techmap.allmem.all;
37 LIBRARY ambarchitect;
38 USE ambarchitect.config.all;
39
40 LIBRARY techmap;
41
42 ARCHITECTURE struct OF mctrl_pad IS
43
44    -- Architecture declarations
45
46    -- Internal signal declarations
47    SIGNAL mctrl_pad_we_n     : std_ulogic;
48    SIGNAL mctrl_pad_wren_map : std_ulogic;
49
50
51    -- Component Declarations
52    COMPONENT iopadv
53    GENERIC (
54       tech     : integer;
55       level    : integer;
56       slew     : integer;
57       voltage  : integer;
58       strength : integer;
59       width    : integer;
60       oepol    : integer
```

```
   61    );
   62    PORT (
   63       en  : IN     std_ulogic;
   64       i   : IN     std_logic_vector (width-1 DOWNTO 0);
   65       o   : OUT    std_logic_vector (width-1 DOWNTO 0);
   66       pad : INOUT  std_logic_vector (width-1 DOWNTO 0)
   67    );
   68    END COMPONENT;
   69    COMPONENT outpad
   70    GENERIC (
   71       tech     : integer;
   72       level    : integer;
   73       slew     : integer;
   74       voltage  : integer;
   75       strength : integer
   76    );
   77    PORT (
   78       i   : IN     std_ulogic;
   79       pad : OUT    std_ulogic
   80    );
   81    END COMPONENT;
   82    COMPONENT outpadv
   83    GENERIC (
   84       tech     : integer;
   85       level    : integer;
   86       slew     : integer;
   87       voltage  : integer;
   88       strength : integer;
   89       width    : integer
   90    );
   91    PORT (
   92       i   : IN     std_logic_vector (width-1 DOWNTO 0);
   93       pad : OUT    std_logic_vector (width-1 DOWNTO 0)
   94    );
   95    END COMPONENT;
   96
   97    -- Optional embedded configurations
   98    -- pragma synthesis_off
   99    FOR ALL : iopadv USE ENTITY techmap.iopadv;
  100    FOR ALL : outpad USE ENTITY techmap.outpad;
  101    FOR ALL : outpadv USE ENTITY techmap.outpadv;
  102    -- pragma synthesis_on
  103
  104
  105 BEGIN
  106    -- Architecture concurrent statements
  107    -- HDL Embedded Text Block 2 eb2
  108    --mctrl_pad_we_n <= NOT (NOT mctrl_pad_sd_we_n
  109    --   AND NOT mctrl_pad_sd_cs_n)
  110    --   OR (NOT mctrl_pad_flash_we_n
  111    --   AND NOT mctrl_pad_flash_cs_n);
  112
  113    mctrl_pad_we_n <= NOT (NOT mctrl_pad_sd_we_n OR NOT
     mctrl_pad_flash_we_n);
  114
  115    --pad enable = 1 >> pad -> o; pad enable = 0 >> i -> pad
  116    --mctrl_pad_wren_map <= '1' WHEN ( (NOT (mctrl_pad_flash_oe_n) =
     '1') OR ( NOT (mctrl_pad_sd_we_n) = '0') ) ELSE '0';
  117    mctrl_pad_wren_map <= '0' WHEN
  118      ((NOT (mctrl_pad_flash_we_n) = '1' AND NOT(mctrl_pad_flash_cs_n)
```

```
      = '1')
  119        OR
  120         ((NOT (mctrl_pad_sd_ras_n) = '0') AND
  121         (NOT (mctrl_pad_sd_we_n) = '1')AND
  122         (NOT (mctrl_pad_sd_cs_n) = '1') AND
  123         (NOT (mctrl_pad_sd_cas_n) = '1') ))
  124       ELSE '1';
  125
  126
  127
  128
  129
  130
  131     -- Instance port mappings.
  132     I0 : iopadv
  133        GENERIC MAP (
  134           tech     => pad_tech,
  135           level    => 1,
  136           slew     => 0,
  137           voltage  => x33v,
  138           strength => 12,
  139           width    => nbDBitsFlash,
  140           oepol    => 0
  141        )
  142        PORT MAP (
  143           pad => data,
  144           i   => mctrl_pad_data_out,
  145           en  => mctrl_pad_wren_map,
  146           o   => mctrl_pad_data_in
  147        );
  148     I10 : outpad
  149        GENERIC MAP (
  150           tech     => pad_tech,
  151           level    => 0,
  152           slew     => 0,
  153           voltage  => x33v,
  154           strength => 12
  155        )
  156        PORT MAP (
  157           pad => flash_cs_n,
  158           i   => mctrl_pad_flash_cs_n
  159        );
  160     I11 : outpad
  161        GENERIC MAP (
  162           tech     => pad_tech,
  163           level    => 0,
  164           slew     => 0,
  165           voltage  => x33v,
  166           strength => 12
  167        )
  168        PORT MAP (
  169           pad => flash_oe_n,
  170           i   => mctrl_pad_flash_oe_n
  171        );
  172     I12 : outpad
  173        GENERIC MAP (
  174           tech     => pad_tech,
  175           level    => 0,
  176           slew     => 0,
  177           voltage  => x33v,
```

```
178           strength => 12
179        )
180        PORT MAP (
181           pad => mem_we_n,
182           i   => mctrl_pad_we_n
183        );
184     I13 : outpad
185        GENERIC MAP (
186           tech     => pad_tech,
187           level    => 0,
188           slew     => 0,
189           voltage  => x33v,
190           strength => 12
191        )
192        PORT MAP (
193           pad => sd_cas_n,
194           i   => mctrl_pad_sd_cas_n
195        );
196     I14 : outpad
197        GENERIC MAP (
198           tech     => pad_tech,
199           level    => 0,
200           slew     => 0,
201           voltage  => x33v,
202           strength => 12
203        )
204        PORT MAP (
205           pad => sd_ras_n,
206           i   => mctrl_pad_sd_ras_n
207        );
208     I15 : outpad
209        GENERIC MAP (
210           tech     => pad_tech,
211           level    => 0,
212           slew     => 0,
213           voltage  => x33v,
214           strength => 12
215        )
216        PORT MAP (
217           pad => sd_cke,
218           i   => mctrl_pad_sd_cke
219        );
220     I16 : outpad
221        GENERIC MAP (
222           tech     => pad_tech,
223           level    => 0,
224           slew     => 0,
225           voltage  => x33v,
226           strength => 12
227        )
228        PORT MAP (
229           pad => sd_cs_n,
230           i   => mctrl_pad_sd_cs_n
231        );
232     I17 : outpad
233        GENERIC MAP (
234           tech     => pad_tech,
235           level    => 0,
236           slew     => 0,
237           voltage  => x33v,
```

```
238           strength => 12
239        )
240        PORT MAP (
241           pad => sd_dqml,
242           i   => mctrl_pad_sd_dqml
243        );
244     I18 : outpad
245        GENERIC MAP (
246           tech     => pad_tech,
247           level    => 0,
248           slew     => 0,
249           voltage  => x33v,
250           strength => 12
251        )
252        PORT MAP (
253           pad => sd_dqmh,
254           i   => mctrl_pad_sd_dqmh
255        );
256     I19 : outpad
257        GENERIC MAP (
258           tech     => pad_tech,
259           level    => 0,
260           slew     => 0,
261           voltage  => x33v,
262           strength => 12
263        )
264        PORT MAP (
265           pad => flashResetn,
266           i   => mctrl_pad_flashResetn
267        );
268     I1 : outpadv
269        GENERIC MAP (
270           tech     => pad_tech,
271           level    => 0,
272           slew     => 0,
273           voltage  => 0,
274           strength => 12,
275           width    => nbABitsFlash
276        )
277        PORT MAP (
278           pad => addr,
279           i   => mctrl_pad_addr
280        );
281
282  END struct;
```

```
 1 --
 2 -- VHDL Architecture leon_toplevel.grgpio_pad.struct
 3 --
 4 -- Created:
 5 --          by - Thomas.UNKNOWN (WE2778)
 6 --          at - 08:40:44 19.05.2009
 7 --
 8 -- Generated by Mentor Graphics' HDL Designer(TM) 2007.1a (Build 13)
 9 --
10 LIBRARY ieee;
11 USE ieee.std_logic_1164.all;
12 USE ieee.numeric_std.all;
13 LIBRARY grlib;
14 USE grlib.stdlib.all;
15 USE grlib.amba.all;
16 USE grlib.devices.all;
17 LIBRARY std;
18 USE std.textio.all;
19 LIBRARY techmap;
20 USE techmap.gencomp.all;
21 LIBRARY gaisler;
22 USE gaisler.misc.all;
23 USE gaisler.uart.all;
24 USE gaisler.libdcom.all;
25 USE gaisler.memctrl.all;
26 USE gaisler.arith.all;
27 USE gaisler.leon3.all;
28 USE gaisler.libiu.all;
29 USE gaisler.net.all;
30 USE gaisler.ethernet_mac.all;
31 USE gaisler.libcache.all;
32 USE gaisler.libproc3.all;
33 USE grlib.multlib.all;
34 LIBRARY esa;
35 USE esa.memoryctrl.all;
36 USE techmap.allmem.all;
37 LIBRARY ambarchitect;
38 USE ambarchitect.config.all;
39
40
41 ARCHITECTURE struct OF grgpio_pad IS
42
43    -- Architecture declarations
44
45    -- Internal signal declarations
46
47
48
49 BEGIN
50    -- Architecture concurrent statements
51    -- HDL Embedded Text Block 1 grgprio_pad_assign
52    pio_pads : for i in 0 to (gpgpio_bus_width-1) generate
53       pio_pad : iopad generic map (tech => pad_tech)
54       port map (pad => gio1(i), en => grgpio_pad_oen(i), i =>
   grgpio_pad_out(i), o => grgpio_pad_in(i));
55    end generate;
56
57
58
59
```

```
60     -- Instance port mappings.
61
62  END struct;
```

```
 1 --
 2 -- VHDL Architecture leon_toplevel.greth_pad.struct
 3 --
 4 -- Created:
 5 --          by - Thomas.UNKNOWN (WE2778)
 6 --          at - 16:05:22 25.06.2009
 7 --
 8 -- Generated by Mentor Graphics' HDL Designer(TM) 2007.1a (Build 13)
 9 --
10 LIBRARY ieee;
11 USE ieee.std_logic_1164.all;
12 USE ieee.numeric_std.all;
13 LIBRARY grlib;
14 USE grlib.stdlib.all;
15 USE grlib.amba.all;
16 USE grlib.devices.all;
17 LIBRARY std;
18 USE std.textio.all;
19 LIBRARY techmap;
20 USE techmap.gencomp.all;
21 LIBRARY gaisler;
22 USE gaisler.misc.all;
23 USE gaisler.uart.all;
24 USE gaisler.libdcom.all;
25 USE gaisler.memctrl.all;
26 USE gaisler.arith.all;
27 USE gaisler.leon3.all;
28 USE gaisler.libiu.all;
29 USE gaisler.net.all;
30 USE gaisler.ethernet_mac.all;
31 USE gaisler.libcache.all;
32 USE gaisler.libproc3.all;
33 USE grlib.multlib.all;
34 LIBRARY esa;
35 USE esa.memoryctrl.all;
36 USE techmap.allmem.all;
37 LIBRARY ambarchitect;
38 USE ambarchitect.config.all;
39
40 LIBRARY techmap;
41
42 ARCHITECTURE struct OF greth_pad IS
43
44    -- Architecture declarations
45
46    -- Internal signal declarations
47    SIGNAL clkpad_rstn : std_ulogic := '1';
48
49
50    -- Component Declarations
51    COMPONENT clkpad
52    GENERIC (
53       tech    : integer;
54       level   : integer;
55       voltage : integer;
56       arch    : integer;
57       hf      : integer
58    );
59    PORT (
60       pad  : IN     std_ulogic;
```

```
   61       rstn : IN     std_ulogic  := '1';
   62       lock : OUT    std_ulogic;
   63       o    : OUT    std_ulogic
   64    );
   65    END COMPONENT;
   66    COMPONENT inpad
   67    GENERIC (
   68       tech     : integer;
   69       level    : integer;
   70       voltage  : integer;
   71       filter   : integer;
   72       strength : integer
   73    );
   74    PORT (
   75       pad : IN     std_ulogic;
   76       o   : OUT    std_ulogic
   77    );
   78    END COMPONENT;
   79    COMPONENT inpadv
   80    GENERIC (
   81       tech    : integer;
   82       level   : integer;
   83       voltage : integer;
   84       width   : integer
   85    );
   86    PORT (
   87       pad : IN     std_logic_vector (width-1 DOWNTO 0);
   88       o   : OUT    std_logic_vector (width-1 DOWNTO 0)
   89    );
   90    END COMPONENT;
   91    COMPONENT iopad
   92    GENERIC (
   93       tech     : integer;
   94       level    : integer;
   95       slew     : integer;
   96       voltage  : integer;
   97       strength : integer;
   98       oepol    : integer
   99    );
  100    PORT (
  101       en  : IN     std_ulogic;
  102       i   : IN     std_ulogic;
  103       o   : OUT    std_ulogic;
  104       pad : INOUT  std_ulogic
  105    );
  106    END COMPONENT;
  107    COMPONENT outpad
  108    GENERIC (
  109       tech     : integer;
  110       level    : integer;
  111       slew     : integer;
  112       voltage  : integer;
  113       strength : integer
  114    );
  115    PORT (
  116       i   : IN     std_ulogic;
  117       pad : OUT    std_ulogic
  118    );
  119    END COMPONENT;
  120    COMPONENT outpadv
```

```
121     GENERIC (
122        tech     : integer;
123        level    : integer;
124        slew     : integer;
125        voltage  : integer;
126        strength : integer;
127        width    : integer
128     );
129     PORT (
130        i   : IN     std_logic_vector (width-1 DOWNTO 0);
131        pad : OUT    std_logic_vector (width-1 DOWNTO 0)
132     );
133     END COMPONENT;
134
135     -- Optional embedded configurations
136     -- pragma synthesis_off
137     FOR ALL : clkpad USE ENTITY techmap.clkpad;
138     FOR ALL : inpad USE ENTITY techmap.inpad;
139     FOR ALL : inpadv USE ENTITY techmap.inpadv;
140     FOR ALL : iopad USE ENTITY techmap.iopad;
141     FOR ALL : outpad USE ENTITY techmap.outpad;
142     FOR ALL : outpadv USE ENTITY techmap.outpadv;
143     -- pragma synthesis_on
144
145
146  BEGIN
147     -- Architecture concurrent statements
148     -- HDL Embedded Text Block 2 eb2
149     clkpad_rstn <= '1';
150
151
152     -- Instance port mappings.
153     I2 : clkpad
154        GENERIC MAP (
155           tech    => pad_tech,
156           level   => 0,
157           voltage => x33v,
158           arch    => 0,
159           hf      => 0
160        )
161        PORT MAP (
162           pad  => eth_tx_clk,
163           o    => eth_pad_tx_clk,
164           rstn => clkpad_rstn,
165           lock => OPEN
166        );
167     I4 : clkpad
168        GENERIC MAP (
169           tech    => pad_tech,
170           level   => 0,
171           voltage => x33v,
172           arch    => 0,
173           hf      => 0
174        )
175        PORT MAP (
176           pad  => eth_rx_clk,
177           o    => eth_pad_rx_clk,
178           rstn => clkpad_rstn,
179           lock => OPEN
180        );
```

```
181     I14 : inpad
182        GENERIC MAP (
183           tech     => pad_tech,
184           level    => 0,
185           voltage  => x33v,
186           filter   => 0,
187           strength => 0
188        )
189        PORT MAP (
190           pad => eth_col,
191           o   => eth_pad_col
192        );
193     I15 : inpad
194        GENERIC MAP (
195           tech     => pad_tech,
196           level    => 0,
197           voltage  => x33v,
198           filter   => 0,
199           strength => 0
200        )
201        PORT MAP (
202           pad => eth_crs,
203           o   => eth_pad_crs
204        );
205     I16 : inpad
206        GENERIC MAP (
207           tech     => pad_tech,
208           level    => 0,
209           voltage  => x33v,
210           filter   => 0,
211           strength => 0
212        )
213        PORT MAP (
214           pad => eth_rx_er,
215           o   => eth_pad_rx_er
216        );
217     I17 : inpad
218        GENERIC MAP (
219           tech     => pad_tech,
220           level    => 0,
221           voltage  => x33v,
222           filter   => 0,
223           strength => 0
224        )
225        PORT MAP (
226           pad => eth_rx_dv,
227           o   => eth_pad_rx_dv
228        );
229     I1 : inpadv
230        GENERIC MAP (
231           tech    => pad_tech,
232           level   => 0,
233           voltage => 0,
234           width   => eth_bus_width
235        )
236        PORT MAP (
237           pad => eth_rxd,
238           o   => eth_pad_rxd
239        );
240     I3 : iopad
```

```
241        GENERIC MAP (
242           tech     => pad_tech,
243           level    => 0,
244           slew     => 0,
245           voltage  => x33v,
246           strength => 12,
247           oepol    => 0
248        )
249        PORT MAP (
250           pad => eth_mdio,
251           i   => eth_pad_mdio_o,
252           en  => eth_pad_tx_en,
253           o   => eth_pad_mdio_i
254        );
255     I10 : outpad
256        GENERIC MAP (
257           tech     => pad_tech,
258           level    => 0,
259           slew     => 0,
260           voltage  => x33v,
261           strength => 12
262        )
263        PORT MAP (
264           pad => eth_reset,
265           i   => eth_pad_reset
266        );
267     I11 : outpad
268        GENERIC MAP (
269           tech     => pad_tech,
270           level    => 0,
271           slew     => 0,
272           voltage  => x33v,
273           strength => 12
274        )
275        PORT MAP (
276           pad => eth_tx_en,
277           i   => eth_pad_tx_en
278        );
279     I12 : outpad
280        GENERIC MAP (
281           tech     => pad_tech,
282           level    => 0,
283           slew     => 0,
284           voltage  => x33v,
285           strength => 12
286        )
287        PORT MAP (
288           pad => eth_mdc,
289           i   => eth_pad_mdc
290        );
291     I0 : outpadv
292        GENERIC MAP (
293           tech     => pad_tech,
294           level    => 0,
295           slew     => 0,
296           voltage  => 0,
297           strength => 12,
298           width    => eth_bus_width
299        )
300        PORT MAP (
```

```
  301           pad => eth_txd,
  302           i   => eth_pad_txd
  303         );
  304
  305  END struct;
```

```
 1 --
 2 -- VHDL Architecture leon_toplevel.dsu_pad.struct
 3 --
 4 -- Created:
 5 --          by - Thomas.UNKNOWN (WE2778)
 6 --          at - 16:05:24 25.06.2009
 7 --
 8 -- Generated by Mentor Graphics' HDL Designer(TM) 2007.1a (Build 13)
 9 --
10 LIBRARY ieee;
11 USE ieee.std_logic_1164.all;
12 USE ieee.numeric_std.all;
13 LIBRARY grlib;
14 USE grlib.stdlib.all;
15 USE grlib.amba.all;
16 USE grlib.devices.all;
17 LIBRARY std;
18 USE std.textio.all;
19 LIBRARY techmap;
20 USE techmap.gencomp.all;
21 LIBRARY gaisler;
22 USE gaisler.misc.all;
23 USE gaisler.uart.all;
24 USE gaisler.libdcom.all;
25 USE gaisler.memctrl.all;
26 USE gaisler.arith.all;
27 USE gaisler.leon3.all;
28 USE gaisler.libiu.all;
29 USE gaisler.net.all;
30 USE gaisler.ethernet_mac.all;
31 USE gaisler.libcache.all;
32 USE gaisler.libproc3.all;
33 USE grlib.multlib.all;
34 LIBRARY esa;
35 USE esa.memoryctrl.all;
36 USE techmap.allmem.all;
37 LIBRARY ambarchitect;
38 USE ambarchitect.config.all;
39 USE techmap.allpads.all;
40
41 LIBRARY techmap;
42
43 ARCHITECTURE struct OF dsu_pad IS
44
45    -- Architecture declarations
46
47    -- Internal signal declarations
48
49
50    -- Component Declarations
51    COMPONENT inpad
52    GENERIC (
53       tech     : integer;
54       level    : integer;
55       voltage  : integer;
56       filter   : integer;
57       strength : integer
58    );
59    PORT (
60       pad : IN     std_ulogic;
```

```
   61       o   : OUT    std_ulogic
   62    );
   63    END COMPONENT;
   64    COMPONENT outpad
   65    GENERIC (
   66       tech     : integer;
   67       level    : integer;
   68       slew     : integer;
   69       voltage  : integer;
   70       strength : integer
   71    );
   72    PORT (
   73       i   : IN     std_ulogic;
   74       pad : OUT    std_ulogic
   75    );
   76    END COMPONENT;
   77
   78    -- Optional embedded configurations
   79    -- pragma synthesis_off
   80    FOR ALL : inpad USE ENTITY techmap.inpad;
   81    FOR ALL : outpad USE ENTITY techmap.outpad;
   82    -- pragma synthesis_on
   83
   84
   85 BEGIN
   86
   87    -- Instance port mappings.
   88    I0 : inpad
   89       GENERIC MAP (
   90          tech     => pad_tech,
   91          level    => 0,
   92          voltage  => x33v,
   93          filter   => 0,
   94          strength => 0
   95       )
   96       PORT MAP (
   97          pad => dsu_enable,
   98          o   => dsu_pad_enable
   99       );
  100    I2 : inpad
  101       GENERIC MAP (
  102          tech     => pad_tech,
  103          level    => 0,
  104          voltage  => x33v,
  105          filter   => 0,
  106          strength => 0
  107       )
  108       PORT MAP (
  109          pad => dsu_break,
  110          o   => dsu_pad_break
  111       );
  112    I1 : outpad
  113       GENERIC MAP (
  114          tech     => pad_tech,
  115          level    => 0,
  116          slew     => 0,
  117          voltage  => x33v,
  118          strength => 12
  119       )
  120       PORT MAP (
```

```
  121           pad => dsu_active,
  122           i   => dsu_pad_active
  123       );
  124
  125 END struct;
```

```
 1 --
 2 -- VHDL Architecture leon_toplevel.clk_pad.struct
 3 --
 4 -- Created:
 5 --          by - Thomas.UNKNOWN (WE2778)
 6 --          at - 15:55:23 26.06.2009
 7 --
 8 -- Generated by Mentor Graphics' HDL Designer(TM) 2007.1a (Build 13)
 9 --
10 LIBRARY ieee;
11 USE ieee.std_logic_1164.all;
12 USE ieee.numeric_std.all;
13 LIBRARY grlib;
14 USE grlib.stdlib.all;
15 USE grlib.amba.all;
16 USE grlib.devices.all;
17 LIBRARY std;
18 USE std.textio.all;
19 LIBRARY techmap;
20 USE techmap.gencomp.all;
21 LIBRARY gaisler;
22 USE gaisler.misc.all;
23 USE gaisler.uart.all;
24 USE gaisler.libdcom.all;
25 USE gaisler.memctrl.all;
26 USE gaisler.arith.all;
27 USE gaisler.leon3.all;
28 USE gaisler.libiu.all;
29 USE gaisler.net.all;
30 USE gaisler.ethernet_mac.all;
31 USE gaisler.libcache.all;
32 USE gaisler.libproc3.all;
33 USE grlib.multlib.all;
34 LIBRARY esa;
35 USE esa.memoryctrl.all;
36 USE techmap.allmem.all;
37 LIBRARY ambarchitect;
38 USE ambarchitect.config.all;
39 USE techmap.allclkgen.all;
40 USE techmap.allpads.all;
41
42 LIBRARY techmap;
43
44 ARCHITECTURE struct OF clk_pad IS
45
46    -- Architecture declarations
47
48    -- Internal signal declarations
49    SIGNAL cgi              : clkgen_in_type;
50    SIGNAL clk_ext_pad_rstn : std_ulogic := '1';
51    SIGNAL pciclkin         : std_logic;
52
53    -- Implicit buffer signal declarations
54    SIGNAL clk_internal : std_ulogic;
55
56
57    -- Component Declarations
58    COMPONENT clkgen
59    GENERIC (
60       tech      : integer;
```

```
   61       clk_mul   : integer;
   62       clk_div   : integer;
   63       sdramen   : integer;
   64       noclkfb   : integer;
   65       pcien     : integer;
   66       pcidll    : integer;
   67       pcisysclk : integer;
   68       freq      : integer;      -- clock frequency in KHz
   69       clk2xen   : integer;
   70       clksel    : integer;      -- enable clock select
   71       clk_odiv  : integer
   72    );
   73    PORT (
   74       cgi      : IN     clkgen_in_type;
   75       clkin    : IN     std_logic;
   76       pciclkin : IN     std_logic;
   77       cgo      : OUT    clkgen_out_type;
   78       clk      : OUT    std_logic;
   79       clk1xu   : OUT    std_logic;
   80       clk2x    : OUT    std_logic;
   81       clk2xu   : OUT    std_logic;
   82       clk4x    : OUT    std_logic;
   83       clkn     : OUT    std_logic;
   84       pciclk   : OUT    std_logic;
   85       sdclk    : OUT    std_logic
   86    );
   87    END COMPONENT;
   88    COMPONENT clkpad
   89    GENERIC (
   90       tech    : integer;
   91       level   : integer;
   92       voltage : integer;
   93       arch    : integer;
   94       hf      : integer
   95    );
   96    PORT (
   97       pad  : IN     std_ulogic;
   98       rstn : IN     std_ulogic  := '1';
   99       lock : OUT    std_ulogic;
  100       o    : OUT    std_ulogic
  101    );
  102    END COMPONENT;
  103
  104    -- Optional embedded configurations
  105    -- pragma synthesis_off
  106    FOR ALL : clkgen USE ENTITY techmap.clkgen;
  107    FOR I10 : clkpad USE ENTITY techmap.clkpad;
  108    -- pragma synthesis_on
  109
  110
  111 BEGIN
  112    -- Architecture concurrent statements
  113    -- HDL Embedded Text Block 2 eb2
  114    clk_ext_pad_rstn <= '1';
  115    sd_clk <= clk_internal;
  116
  117    pciclkin <= '0';
  118    cgi.pllref <= '0';
  119    cgi.pllrst <= '0';
  120    cgi.pllctrl <= (others => '0');
```

```
121     cgi.clksel <= (others => '0');
122
123
124
125
126     -- Instance port mappings.
127     I0 : clkgen
128        GENERIC MAP (
129           tech      => pad_tech,
130           clk_mul   => 1,
131           clk_div   => 1,
132           sdramen   => 1,
133           noclkfb   => 1,
134           pcien     => 0,
135           pcidll    => 0,
136           pcisysclk => 0,
137           freq      => 25000,         -- clock frequency in KHz
138           clk2xen   => 1,
139           clksel    => 0,             -- enable clock select
140           clk_odiv  => 0
141        )
142        PORT MAP (
143           clkin    => clk_internal,
144           pciclkin => pciclkin,
145           clk      => OPEN,
146           clkn     => OPEN,
147           clk2x    => OPEN,
148           sdclk    => OPEN,
149           pciclk   => OPEN,
150           cgi      => cgi,
151           cgo      => OPEN,
152           clk4x    => OPEN,
153           clk1xu   => OPEN,
154           clk2xu   => OPEN
155        );
156     I10 : clkpad
157        GENERIC MAP (
158           tech    => pad_tech,
159           level   => 0,
160           voltage => x33v,
161           arch    => 0,
162           hf      => 0
163        )
164        PORT MAP (
165           pad  => clk_ext,
166           o    => clk_internal,
167           rstn => clk_ext_pad_rstn,
168           lock => OPEN
169        );
170
171     g0: IF false GENERATE
172     -- Optional embedded configurations
173     -- pragma synthesis_off
174     FOR I1 : clkpad USE ENTITY techmap.clkpad;
175     -- pragma synthesis_on
176
177     BEGIN
178        I1 : clkpad
179           GENERIC MAP (
180              tech    => pad_tech,
```

```
181             level   => 0,
182             voltage => x33v,
183             arch    => 0,
184             hf      => 0
185          )
186          PORT MAP (
187             pad  => clk_internal,
188             o    => OPEN,
189             rstn => clk_ext_pad_rstn,
190             lock => OPEN
191          );
192    END GENERATE g0;
193
194    -- Implicit buffered output assignments
195    clk <= clk_internal;
196
197 END struct;
```

```
 1 --
 2 -- VHDL Architecture leon_toplevel.apbuart_pad.struct
 3 --
 4 -- Created:
 5 --          by - Thomas.UNKNOWN (WE2778)
 6 --          at - 16:05:20 25.06.2009
 7 --
 8 -- Generated by Mentor Graphics' HDL Designer(TM) 2007.1a (Build 13)
 9 --
10 LIBRARY ieee;
11 USE ieee.std_logic_1164.all;
12 USE ieee.numeric_std.all;
13 LIBRARY grlib;
14 USE grlib.stdlib.all;
15 USE grlib.amba.all;
16 USE grlib.devices.all;
17 LIBRARY std;
18 USE std.textio.all;
19 LIBRARY techmap;
20 USE techmap.gencomp.all;
21 LIBRARY gaisler;
22 USE gaisler.misc.all;
23 USE gaisler.uart.all;
24 USE gaisler.libdcom.all;
25 USE gaisler.memctrl.all;
26 USE gaisler.arith.all;
27 USE gaisler.leon3.all;
28 USE gaisler.libiu.all;
29 USE gaisler.net.all;
30 USE gaisler.ethernet_mac.all;
31 USE gaisler.libcache.all;
32 USE gaisler.libproc3.all;
33 USE grlib.multlib.all;
34 LIBRARY esa;
35 USE esa.memoryctrl.all;
36 USE techmap.allmem.all;
37 LIBRARY ambarchitect;
38 USE ambarchitect.config.all;
39
40 LIBRARY techmap;
41
42 ARCHITECTURE struct OF apbuart_pad IS
43
44    -- Architecture declarations
45
46    -- Internal signal declarations
47
48
49    -- Component Declarations
50    COMPONENT inpad
51    GENERIC (
52       tech     : integer;
53       level    : integer;
54       voltage  : integer;
55       filter   : integer;
56       strength : integer
57    );
58    PORT (
59       pad : IN     std_ulogic;
60       o   : OUT    std_ulogic
```

```
 61     );
 62     END COMPONENT;
 63     COMPONENT outpad
 64     GENERIC (
 65        tech     : integer;
 66        level    : integer;
 67        slew     : integer;
 68        voltage  : integer;
 69        strength : integer
 70     );
 71     PORT (
 72        i   : IN     std_ulogic;
 73        pad : OUT    std_ulogic
 74     );
 75     END COMPONENT;
 76
 77     -- Optional embedded configurations
 78     -- pragma synthesis_off
 79     FOR ALL : inpad USE ENTITY techmap.inpad;
 80     FOR ALL : outpad USE ENTITY techmap.outpad;
 81     -- pragma synthesis_on
 82
 83
 84  BEGIN
 85
 86     -- Instance port mappings.
 87     I5 : inpad
 88        GENERIC MAP (
 89           tech     => pad_tech,
 90           level    => 0,
 91           voltage  => x33v,
 92           filter   => 0,
 93           strength => 0
 94        )
 95        PORT MAP (
 96           pad => rs232_rx1,
 97           o   => apbuart_rx
 98        );
 99     I7 : outpad
100        GENERIC MAP (
101           tech     => pad_tech,
102           level    => 0,
103           slew     => 0,
104           voltage  => x33v,
105           strength => 12
106        )
107        PORT MAP (
108           pad => rs232_tx1,
109           i   => apbuart_tx
110        );
111
112  END struct;
```

```
 1 --
 2 -- VHDL Architecture leon_toplevel.ahbuart_pad.struct
 3 --
 4 -- Created:
 5 --          by - Thomas.UNKNOWN (WE2778)
 6 --          at - 16:05:23 25.06.2009
 7 --
 8 -- Generated by Mentor Graphics' HDL Designer(TM) 2007.1a (Build 13)
 9 --
10 LIBRARY ieee;
11 USE ieee.std_logic_1164.all;
12 USE ieee.numeric_std.all;
13 LIBRARY grlib;
14 USE grlib.stdlib.all;
15 USE grlib.amba.all;
16 USE grlib.devices.all;
17 LIBRARY std;
18 USE std.textio.all;
19 LIBRARY techmap;
20 USE techmap.gencomp.all;
21 LIBRARY gaisler;
22 USE gaisler.misc.all;
23 USE gaisler.uart.all;
24 USE gaisler.libdcom.all;
25 USE gaisler.memctrl.all;
26 USE gaisler.arith.all;
27 USE gaisler.leon3.all;
28 USE gaisler.libiu.all;
29 USE gaisler.net.all;
30 USE gaisler.ethernet_mac.all;
31 USE gaisler.libcache.all;
32 USE gaisler.libproc3.all;
33 USE grlib.multlib.all;
34 LIBRARY esa;
35 USE esa.memoryctrl.all;
36 USE techmap.allmem.all;
37 LIBRARY ambarchitect;
38 USE ambarchitect.config.all;
39
40 LIBRARY techmap;
41
42 ARCHITECTURE struct OF ahbuart_pad IS
43
44    -- Architecture declarations
45
46    -- Internal signal declarations
47
48
49    -- Component Declarations
50    COMPONENT inpad
51    GENERIC (
52       tech     : integer;
53       level    : integer;
54       voltage  : integer;
55       filter   : integer;
56       strength : integer
57    );
58    PORT (
59       pad : IN     std_ulogic;
60       o   : OUT    std_ulogic
```

```
61     );
62     END COMPONENT;
63     COMPONENT outpad
64     GENERIC (
65        tech     : integer;
66        level    : integer;
67        slew     : integer;
68        voltage  : integer;
69        strength : integer
70     );
71     PORT (
72        i   : IN     std_ulogic;
73        pad : OUT    std_ulogic
74     );
75     END COMPONENT;
76
77     -- Optional embedded configurations
78     -- pragma synthesis_off
79     FOR ALL : inpad USE ENTITY techmap.inpad;
80     FOR ALL : outpad USE ENTITY techmap.outpad;
81     -- pragma synthesis_on
82
83
84  BEGIN
85
86     -- Instance port mappings.
87     I4 : inpad
88        GENERIC MAP (
89           tech     => pad_tech,
90           level    => 0,
91           voltage  => x33v,
92           filter   => 0,
93           strength => 0
94        )
95        PORT MAP (
96           pad => rs232_rx0,
97           o   => ahbuart_rx
98        );
99     I3 : outpad
100       GENERIC MAP (
101          tech     => pad_tech,
102          level    => 0,
103          slew     => 0,
104          voltage  => x33v,
105          strength => 12
106       )
107       PORT MAP (
108          pad => rs232_tx0,
109          i   => ahbuart_tx
110       );
111
112 END struct;
```

Beilage 16

LEON Schaltung .ucf (I/O Mapping)

```
 1 #
 2 # Constraints generated by Synplify Pro 9.4.0, Build 086R
 3 #
 4
 5 # Period Constraints
 6
 7 #Begin clock constraints
 8 NET "clk_ext" TNM_NET = "clk_ext";
 9 TIMESPEC "TS_clk_ext" = PERIOD "clk_ext" 15.439 ns HIGH 50.00%;
10 #End clock constraints
11
12
13 # I/O Registers Packing Constraints
14 INST "I0/I31/r.romsn_rep0_i[0]" IOB=FALSE;
15 INST "I0/I31/r.data[28]" IOB=FALSE;
16 INST "I0/I31/r.data[27]" IOB=FALSE;
17 INST "I0/I31/r.data[26]" IOB=FALSE;
18 INST "I0/I31/r.data[25]" IOB=FALSE;
19 INST "I0/I31/r.data[24]" IOB=FALSE;
20 INST "I0/I31/r.data[23]" IOB=FALSE;
21 INST "I0/I31/r.data[22]" IOB=FALSE;
22 INST "I0/I31/r.data[21]" IOB=FALSE;
23 INST "I0/I31/r.data[20]" IOB=FALSE;
24 INST "I0/I31/r.data[19]" IOB=FALSE;
25 INST "I0/I31/r.data[18]" IOB=FALSE;
26 INST "I0/I31/r.data[17]" IOB=FALSE;
27 INST "I0/I31/r.data[16]" IOB=FALSE;
28 INST "I0/I31/r.data[31]" IOB=FALSE;
29 INST "I0/I31/r.data[30]" IOB=FALSE;
30 INST "I0/I31/r.data[29]" IOB=FALSE;
31 INST "I0/I31/sdo.dqm[1]" IOB=FALSE;
32 INST "I0/I31/sdo.dqm[0]" IOB=FALSE;
33 INST "I0/I31/sdo.sdcke[1]" IOB=FALSE;
34
35 # I/O Registers Packing Constraints
36 INST "I0/I8/r.txd" IOB=FALSE;
37
38 # I/O Registers Packing Constraints
39 INST "I0/I18/r.dir_rep0[16]" IOB=FALSE;
40 INST "I0/I18/r.dir_rep0[15]" IOB=FALSE;
41 INST "I0/I18/r.dir_rep0[14]" IOB=FALSE;
42 INST "I0/I18/r.dir_rep0[13]" IOB=FALSE;
43 INST "I0/I18/r.dir_rep0[12]" IOB=FALSE;
44 INST "I0/I18/r.dir_rep0[11]" IOB=FALSE;
45 INST "I0/I18/r.dir_rep0[10]" IOB=FALSE;
46 INST "I0/I18/r.dir_rep0[9]" IOB=FALSE;
47 INST "I0/I18/r.dir_rep0[8]" IOB=FALSE;
48 INST "I0/I18/r.dir_rep0[7]" IOB=FALSE;
49 INST "I0/I18/r.dir_rep0[6]" IOB=FALSE;
50 INST "I0/I18/r.dir_rep0[5]" IOB=FALSE;
51 INST "I0/I18/r.dir_rep0[4]" IOB=FALSE;
52 INST "I0/I18/r.dir_rep0[3]" IOB=FALSE;
53 INST "I0/I18/r.dir_rep0[2]" IOB=FALSE;
54 INST "I0/I18/r.dir_rep0[1]" IOB=FALSE;
55 INST "I0/I18/r.dir_rep0[0]" IOB=FALSE;
56
57 # I/O Registers Packing Constraints
58 INST "I7/Q" IOB=FALSE;
59
60 # I/O Registers Packing Constraints
```

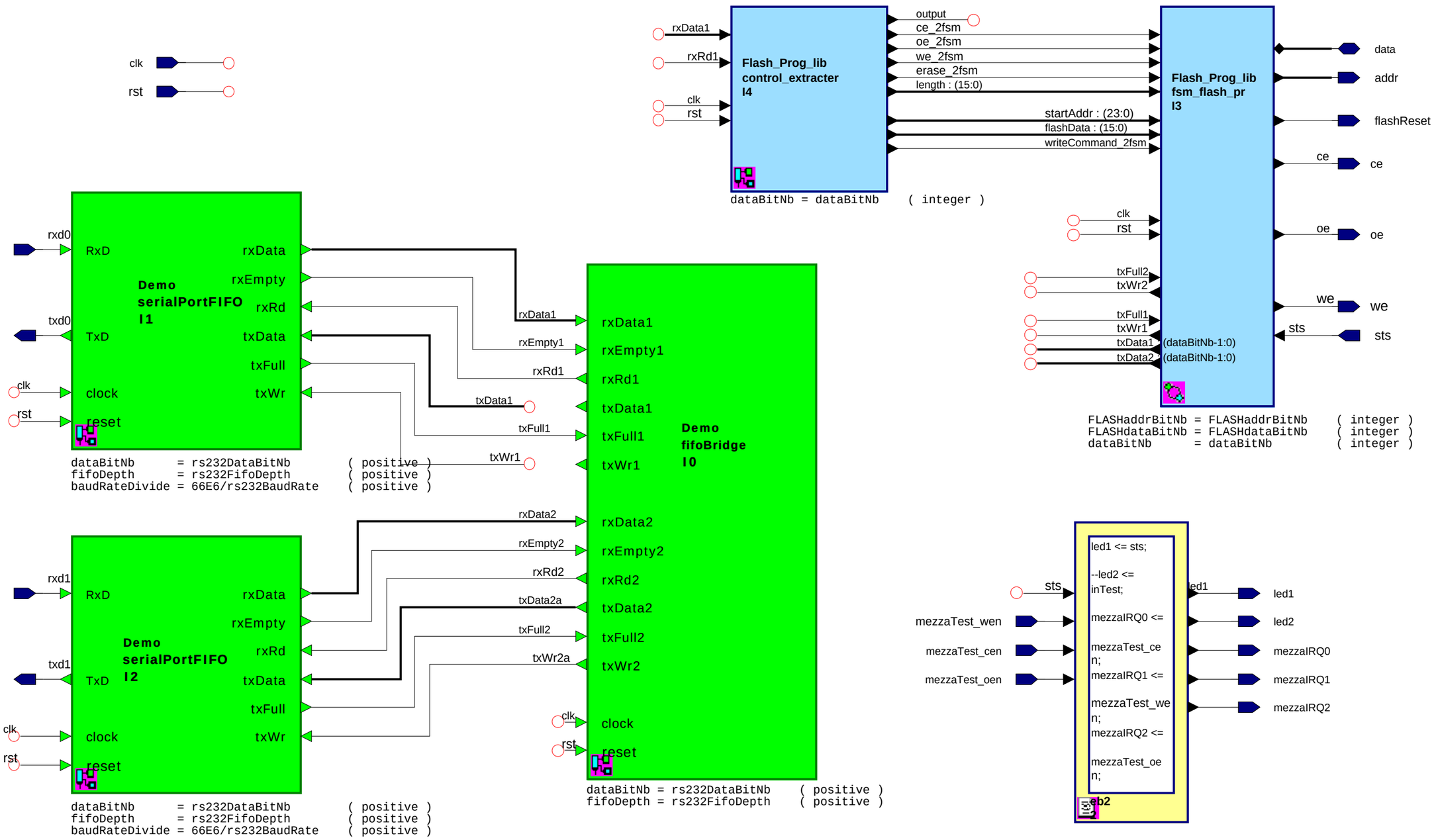
```
 61  INST "I10/Q" IOB=FALSE;
 62
 63  # I/O Registers Packing Constraints
 64  INST "I11/Q" IOB=FALSE;
 65
 66  # I/O Registers Packing Constraints
 67  INST "I12/Q" IOB=FALSE;
 68
 69  # I/O Registers Packing Constraints
 70  INST "I14/Q[5]" IOB=FALSE;
 71  INST "I14/Q[4]" IOB=FALSE;
 72  INST "I14/Q[3]" IOB=FALSE;
 73  INST "I14/Q[2]" IOB=FALSE;
 74  INST "I14/Q[1]" IOB=FALSE;
 75  INST "I14/Q[0]" IOB=FALSE;
 76  INST "I14/Q[16]" IOB=FALSE;
 77  INST "I14/Q[15]" IOB=FALSE;
 78  INST "I14/Q[14]" IOB=FALSE;
 79  INST "I14/Q[13]" IOB=FALSE;
 80  INST "I14/Q[12]" IOB=FALSE;
 81  INST "I14/Q[11]" IOB=FALSE;
 82  INST "I14/Q[10]" IOB=FALSE;
 83  INST "I14/Q[9]" IOB=FALSE;
 84  INST "I14/Q[8]" IOB=FALSE;
 85  INST "I14/Q[7]" IOB=FALSE;
 86  INST "I14/Q[6]" IOB=FALSE;
 87
 88  # End of generated constraints
 89  NET "eth_col"  LOC = "T2"  ;
 90  NET "eth_crs"  LOC = "R2"  ;
 91  NET "eth_rx_clk"  LOC = "C9"  ;
 92  NET "eth_rx_dv"  LOC = "V4"  ;
 93  NET "eth_rx_er"  LOC = "V3"  ;
 94  NET "eth_rxd(0)"  LOC = "U13"  ;
 95  NET "eth_rxd(1)"  LOC = "L13"  ;
 96  NET "eth_rxd(2)"  LOC = "L14"  ;
 97  NET "eth_rxd(3)"  LOC = "U14"  ;
 98  NET "eth_tx_clk"  LOC = "D9"  ;
 99  #PACE: Start of Constraints generated by PACE
100
101  #PACE: Start of PACE I/O Pin Assignments
102  NET "addr(0)"  LOC = "P18"  ;
103  NET "addr(1)"  LOC = "J13"  ;
104  NET "addr(10)"  LOC = "K13"  ;
105  NET "addr(11)"  LOC = "L15"  ;
106  NET "addr(12)"  LOC = "L16"  ;
107  NET "addr(13)"  LOC = "T18"  ;
108  NET "addr(14)"  LOC = "R18"  ;
109  NET "addr(15)"  LOC = "T17"  ;
110  NET "addr(16)"  LOC = "U18"  ;
111  NET "addr(17)"  LOC = "T16"  ;
112  NET "addr(18)"  LOC = "U15"  ;
113  NET "addr(19)"  LOC = "V15"  ;
114  NET "addr(2)"  LOC = "J12"  ;
115  NET "addr(20)"  LOC = "R14"  ;
116  NET "addr(21)"  LOC = "T14"  ;
117  NET "addr(22)"  LOC = "R13"  ;
118  NET "addr(23)"  LOC = "P13"  ;
119  NET "addr(3)"  LOC = "J14"  ;
120  NET "addr(4)"  LOC = "J15"  ;
```

```
121  NET "addr(5)"  LOC = "J16"  ;
122  NET "addr(6)"  LOC = "J17"  ;
123  NET "addr(7)"  LOC = "K14"  ;
124  NET "addr(8)"  LOC = "K15"  ;
125  NET "addr(9)"  LOC = "K12"  ;
126  NET "clk_ext"  LOC = "A10"  ;
127  NET "data(0)"  LOC = "H17"  ;
128  NET "data(1)"  LOC = "F17"  ;
129  NET "data(10)"  LOC = "C18"  ;
130  NET "data(11)"  LOC = "G14"  ;
131  NET "data(12)"  LOC = "F14"  ;
132  NET "data(13)"  LOC = "D11"  ;
133  NET "data(14)"  LOC = "F15"  ;
134  NET "data(15)"  LOC = "D16"  ;
135  NET "data(2)"  LOC = "D17"  ;
136  NET "data(3)"  LOC = "C17"  ;
137  NET "data(4)"  LOC = "G15"  ;
138  NET "data(5)"  LOC = "E11"  ;
139  NET "data(6)"  LOC = "G16"  ;
140  NET "data(7)"  LOC = "E13"  ;
141  NET "data(8)"  LOC = "F18"  ;
142  NET "data(9)"  LOC = "E17"  ;
143  NET "dsu_active"  LOC = "A16"  ;
144  NET "dsu_enable"  LOC = "D3" | PULLUP ;
145  NET "eth_mdc"  LOC = "N5"  ;
146  NET "eth_mdio"  LOC = "R5"  ;
147  NET "eth_reset"  LOC = "P2"  ;
148  NET "eth_tx_en"  LOC = "N4"  ;
149  NET "eth_txd(0)"  LOC = "M4"  ;
150  NET "eth_txd(1)"  LOC = "U3"  ;
151  NET "eth_txd(2)"  LOC = "R3"  ;
152  NET "eth_txd(3)"  LOC = "M3"  ;
153  NET "flash_cs_n"  LOC = "M16"  ;
154  NET "flash_oe_n"  LOC = "H15"  ;
155  NET "flashResetn"  LOC = "M18"  ;
156  NET "gio1(0)"  LOC = "G4"  ;
157  NET "gio1(1)"  LOC = "G5"  ;
158  NET "gio1(10)"  LOC = "C5"  ;
159  NET "gio1(11)"  LOC = "G6"  ;
160  NET "gio1(12)"  LOC = "B3"  ;
161  NET "gio1(13)"  LOC = "B4"  ;
162  NET "gio1(14)"  LOC = "A4"  ;
163  NET "gio1(15)"  LOC = "B6"  ;
164  NET "gio1(2)"  LOC = "E2"  ;
165  NET "gio1(3)"  LOC = "D2"  ;
166  NET "gio1(4)"  LOC = "G3"  ;
167  NET "gio1(5)"  LOC = "F4"  ;
168  NET "gio1(6)"  LOC = "D5"  ;
169  NET "gio1(7)"  LOC = "C2"  ;
170  NET "gio1(8)"  LOC = "C3"  ;
171  NET "gio1(9)"  LOC = "C4"  ;
172  NET "mem_we_n"  LOC = "H16"  ;
173  NET "rs232_rx0"  LOC = "V2"  ;
174  NET "rs232_rx1"  LOC = "U1"  ;
175  NET "rs232_tx0"  LOC = "T1"  ;
176  NET "rs232_tx1"  LOC = "P1"  ;
177  NET "rstn"  LOC = "A15" | PULLUP ;
178  NET "sd_cas_n"  LOC = "M14"  ;
179  NET "sd_cke"  LOC = "P15"  ;
180  NET "sd_clk"  LOC = "R16" | IOSTANDARD = LVTTL ;
```

```
181 NET "sd_cs_n"  LOC = "M13"  ;
182 NET "sd_dqmh"  LOC = "R15"  ;
183 NET "sd_dqml"  LOC = "T15"  ;
184 NET "sd_ras_n"  LOC = "V13"  ;
185
186 #PACE: Start of PACE Area Constraints
187
188 #PACE: Start of PACE Prohibit Constraints
189
190 #PACE: End of Constraints generated by PACE
```

Beilage 17

FLASH Programmation Toplevel


clk
rst
rxd0
RxD
txd0
TxD
clk
rst
clock
reset
Demo
serialPortFIFO
I1
rxData
rxEmpty
rxRd
txData
txFull
txWr
dataBitNb = rs232DataBitNb (positive)
fifoDepth = rs232FifoDepth (positive)
baudRateDivide = 66E6/rs232BaudRate (positive)
rxData1
rxEmpty1
rxRd1
txData1
txFull1
txWr1
rxData1
rxRd1
clk
rst
Flash_Prog_lib
control_extracter
I4
output
ce_2fsm
oe_2fsm
we_2fsm
erase_2fsm
length : (15:0)
startAddr : (23:0)
flashData : (15:0)
writeCommand_2fsm
dataBitNb = dataBitNb (integer)
clk
rst
txFull2
txWr2
txFull1
txWr1
txData1
txData2
Flash_Prog_lib
fsm_flash_pr
I3
(dataBitNb-1:0)
(dataBitNb-1:0)
data
addr
flashReset
ce
ce
oe
oe
we
we
sts
sts
FLASHaddrBitNb = FLASHaddrBitNb (integer)
FLASHdataBitNb = FLASHdataBitNb (integer)
dataBitNb = dataBitNb (integer)
rxData1
rxEmpty1
rxRd1
txData1
txFull1
txWr1
rxData2
rxEmpty2
rxRd2
txData2a
txFull2
txWr2a
clk
rst
Demo
fifoBridge
I0
rxData2
rxEmpty2
rxRd2
txData2
txFull2
txWr2
clock
reset
dataBitNb = rs232DataBitNb (positive)
fifoDepth = rs232FifoDepth (positive)
rxd1
RxD
txd1
TxD
clk
rst
clock
reset
Demo
serialPortFIFO
I2
rxData
rxEmpty
rxRd
txData
txFull
txWr
dataBitNb = rs232DataBitNb (positive)
fifoDepth = rs232FifoDepth (positive)
baudRateDivide = 66E6/rs232BaudRate (positive)
sts
mezzaTest_wen
mezzaTest_cen
mezzaTest_oen
led1 <= sts;
--led2 <= inTest;
mezzaIRQ0 <= mezzaTest_cen;
mezzaIRQ1 <= mezzaTest_wen;
mezzaIRQ2 <= mezzaTest_oen;
eb2
led1
led2
mezzaIRQ0
mezzaIRQ1
mezzaIRQ2

Beilage 18

FLASH Control Extractor VHDL Architektur

```
 1 --
 2 -- VHDL Architecture Flash_Prog_lib.control_extracter.struct
 3 --
 4 -- Created:
 5 --          by - Thomas.UNKNOWN (WE2778)
 6 --          at - 14:50:49 17.06.2009
 7 --
 8 -- Generated by Mentor Graphics' HDL Designer(TM) 2007.1a (Build 13)
 9 --
10 LIBRARY ieee;
11 USE ieee.std_logic_1164.all;
12 USE ieee.numeric_std.all;
13
14
15 ARCHITECTURE struct OF control_extracter IS
16
17    -- Architecture declarations
18    constant CB: std_ulogic_vector(dataBitNb-1 DOWNTO 0) :=
   "11001011";
19    constant HEADER101: std_ulogic_vector(2 DOWNTO 0) := "101";
20    constant ERASEBIT: positive := 4;
21    constant CEBIT: positive := 3;
22    constant OEBIT: positive := 2;
23    constant WEBIT: positive := 1;
24    constant STSBIT: natural := 0;
25
26    -- Internal signal declarations
27    SIGNAL counter          : unsigned(1 DOWNTO 0);
28    SIGNAL counter6         : unsigned(2 DOWNTO 0);
29    SIGNAL counterOF        : std_ulogic;
30    SIGNAL counterOF6       : std_ulogic;
31    SIGNAL eraseCommand     : std_ulogic;
32    SIGNAL isHex            : std_ulogic;
33    SIGNAL led1             : std_logic;
34    SIGNAL lengthCommand    : std_ulogic;
35    SIGNAL readCommand      : std_ulogic;
36    SIGNAL rsData           : unsigned(dataBitNb-1 DOWNTO 0);
37    SIGNAL rxFourNibbles    : unsigned(15 DOWNTO 0);
38    SIGNAL rxNibble         : unsigned(3 DOWNTO 0);
39    SIGNAL rxSixNibbles     : unsigned(23 DOWNTO 0);
40    SIGNAL startAddrCommand : std_ulogic;
41    SIGNAL writeCommand     : std_ulogic;
42
43
44
45 BEGIN
46    -- Architecture concurrent statements
47    -- HDL Embedded Text Block 1 eb1
48       -- convert to upper case to make system case independent
49       upperCase: process(rxData1)
50       begin
51         if (unsigned(rxData1) >= character'pos('a')) and
52            (unsigned(rxData1) <= character'pos('z')) then
53            rsData <= unsigned(rxData1) - character'pos('a') +
   character'pos('A');
54         else
55           rsData <= unsigned(rxData1);
56         end if;
57       end process upperCase;
58
```

```
59        --check for allowed chars in HEX range 0-9, A-F
60        checkHex: process(rsData)
61        begin
62          if (rsData >= character'pos('0')) and (rsData <=
    character'pos('9')) then
63            isHex <= '1';
64          elsif (rsData >= character'pos('A')) and (rsData <=
    character'pos('F')) then
65            isHex <= '1';
66          else
67            isHex <= '0';
68          end if;
69        end process checkHex;
70
71        --save nibble for data frames
72        extractNibble: process(rsData)
73        begin
74          if (rsData >= character'pos('0')) and (rsData <=
    character'pos('9')) then
75            rxNibble <= rsData(rxNibble'range);
76          elsif (rsData >= character'pos('A')) and (rsData <=
    character'pos('F')) then
77            rxNibble <= rsData(rxNibble'range) - 1 + 10; -- 'A' = 0x41
    -> 0x40 + 0xA = 0x4A -> A
78          else
79            rxNibble <= (others => '-');
80          end if;
81        end process extractNibble;
82
83        --set internal command signals accoring to received commands
84        storeControls: process(clk, rst)
85        begin
86          if rst = '1' then
87            eraseCommand <= '0';
88            startAddrCommand <= '0';
89            lengthCommand <= '0';
90            writeCommand <= '0';
91            readCommand <= '0';
92          elsif rising_edge(clk) then
93            if rxRd1 = '1' then
94              if rsData = character'pos('X') then
95                --set ERASE COMMAND and clear all others
96                eraseCommand <= '1';
97                startAddrCommand <= '0';
98                lengthCommand <= '0';
99                writeCommand <= '0';
100               readCommand <= '0';
101             elsif rsData = character'pos('S') then
102               --set START ADDR COMMAND and clear all others
103               eraseCommand <= '0';
104               startAddrCommand <= '1';
105               lengthCommand <= '0';
106               writeCommand <= '0';
107               readCommand <= '0';
108             elsif rsData = character'pos('W') then
109               --set WRITE COMMAND and clear all others
110               eraseCommand <= '0';
111               startAddrCommand <= '0';
112               lengthCommand <= '0';
113               writeCommand <= '1';
```

```
114                 readCommand <= '0';
115               elsif rsData = character'pos('L') then
116                 --set LENGTH COMMAND and clear all others
117                 eraseCommand <= '0';
118                 startAddrCommand <= '0';
119                 lengthCommand <= '1';
120                 writeCommand <= '0';
121                 readCommand <= '0';
122               elsif rsData = character'pos('R') then
123                 --set READ COMMAND and clear all others
124                 eraseCommand <= '0';
125                 startAddrCommand <= '0';
126                 lengthCommand <= '0';
127                 writeCommand <= '0';
128                 readCommand <= '1';
129               end if;
130             end if;
131           end if;
132         end process storeControls;
133
134         --send WRITE COMMAND to FSM
135         writeCommand_2fsm <= writeCommand;
136
137
138         --nibble counter 3 -> 0
139         nibbleCounter: process(clk, rst)
140         begin
141           if rst = '1' then
142             counter <= "11";
143           elsif rising_edge(clk) then
144             if rxrd1 = '1' then
145               if isHex = '1' then
146                 --only for LENGTH and WRITE COMMANDS
147                 if lengthCommand = '1' OR writeCommand = '1' then
148                    counter <= counter - 1;
149                 else
150                    counter <= "11";
151                 end if;
152               end if;
153             end if;
154           end if;
155         end process nibbleCounter;
156
157         --nibble counter overflow check
158         nibbleCounterOF: process (clk, rst)
159         begin
160           if rst = '1' then
161             counterOF <= '0';
162           elsif rising_edge(clk) then
163             if counter = 0 AND rxrd1 = '1' then
164               counterOF <= '1';
165             else
166               counterOF <= '0';
167             end if;
168           end if;
169         end process nibbleCounterOF;
170         --nibbleCounterOF: process (rxrd1, counter)
171         --begin
172         --  if counter = 0 AND rxrd1 = '1' then
173         --    counterOF <= '1';
```

```
174       --  else
175       --    counterOF <= '0';
176       --  end if;
177       --end process nibbleCounterOF;
178
179       --nibble counter 5 -> 0
180       nibbleCounter6: process(clk, rst)
181       begin
182         if rst = '1' then
183           counter6 <= "101";
184         elsif rising_edge(clk) then
185           if rxrd1 = '1' then
186             if isHex = '1' then
187               --only for START ADDR COMMAND
188               if startAddrCommand = '1' then
189                 counter6 <= counter6 - 1;
190               else
191                 counter6 <= "101";
192               end if;
193             end if;
194           end if;
195         end if;
196       end process nibbleCounter6;
197
198       --nibble counter overflow check
199       nibbleCounterOF6: process (clk, rst)
200       begin
201         if rst = '1' then
202           counterOF6 <= '0';
203         elsif rising_edge(clk) then
204           if counter6 = 0 AND rxrd1 = '1' then
205             counterOF6 <= '1';
206           else
207             counterOF6 <= '0';
208           end if;
209         end if;
210       end process nibbleCounterOF6;
211       --nibbleCounterOF6: process (rxrd1, counter6)
212       --begin
213       --  if counter6 = 0 AND rxrd1 = '1' then
214       --    counterOF6 <= '1';
215       --  else
216       --    counterOF6 <= '0';
217       --  end if;
218       --end process nibbleCounterOF6;
219
220       --combine the four nibbles in a 16 bit signal
221       setFourNibbles: process(clk, rst)
222       begin
223         if rst = '1' then
224           rxFourNibbles <= (others => '0');
225         elsif rising_edge(clk) then
226           if rxrd1 = '1' then
227             if counter = 3 then
228               rxFourNibbles(rxFourNibbles'high downto
    rxFourNibbles'high-rxNibble'length+1) <= rxNibble;
229             elsif counter = 2 then
230               rxFourNibbles(rxFourNibbles'high-rxNibble'length downto
    rxFourNibbles'high-2*rxNibble'length+1) <= rxNibble;
231             elsif counter = 1 then
```

```
232               rxFourNibbles(rxFourNibbles'high-2*rxNibble'length
    downto rxFourNibbles'high-3*rxNibble'length+1) <= rxNibble;
233             elsif counter = 0 then
234               rxFourNibbles(rxFourNibbles'high-3*rxNibble'length
    downto 0) <= rxNibble;
235             end if;
236           end if;
237         end if;
238       end process setFourNibbles;
239
240       --combine the six nibbles in a 24 bit signal
241       setSixNibbles: process(clk, rst)
242       begin
243         if rst = '1' then
244           rxSixNibbles <= (others => '0');
245         elsif rising_edge(clk) then
246           if rxrd1 = '1' then
247             if counter6 = 5 then
248               rxSixNibbles(rxSixNibbles'high downto
    rxSixNibbles'high-rxNibble'length+1) <= rxNibble;
249             elsif counter6 = 4 then
250               rxSixNibbles(rxSixNibbles'high-rxNibble'length downto
    rxSixNibbles'high-2*rxNibble'length+1) <= rxNibble;
251             elsif counter6 = 3 then
252               rxSixNibbles(rxSixNibbles'high-2*rxNibble'length downto
    rxSixNibbles'high-3*rxNibble'length+1) <= rxNibble;
253             elsif counter6 = 2 then
254               rxSixNibbles(rxSixNibbles'high-3*rxNibble'length downto
    rxSixNibbles'high-4*rxNibble'length+1) <= rxNibble;
255             elsif counter6 = 1 then
256               rxSixNibbles(rxSixNibbles'high-4*rxNibble'length downto
    rxSixNibbles'high-5*rxNibble'length+1) <= rxNibble;
257             elsif counter6 = 0 then
258               rxSixNibbles(rxSixNibbles'high-5*rxNibble'length downto
    0) <= rxNibble;
259             end if;
260           end if;
261         end if;
262       end process setSixNibbles;
263
264
265       --set registers or output data for FSM (16 bit signals)
266       setOutputs: process (clk, rst)
267       begin
268         if rst = '1' then
269           length <= (others => '0');
270           flashData <= rxFourNibbles;
271           we_2fsm <= '0';
272         elsif rising_edge(clk) then
273           we_2fsm <= '0';
274           if counterOF = '1' then
275             if lengthCommand = '1' then
276               --forward length value to FSM
277               length <= rxFourNibbles;
278             elsif writeCommand = '1' then
279               --forward data for flash to FSM
280               flashData <= rxFourNibbles;
281               --indicate write operation to FSM
282               we_2fsm <= '1';
283             end if;
```

```
284           end if;
285         end if;
286       end process setOutputs;
287
288       --setOutputs: process (counterOF, lengthCommand, rxFourNibbles,
    writeCommand)
289       --begin
290       --  we_2fsm <= '0';
291       --  if counterOF = '1' then
292       --    if lengthCommand = '1' then
293       --      --forward length value to FSM
294       --      length <= rxFourNibbles;
295       --    elsif writeCommand = '1' then
296       --      --forward data for flash to FSM
297       --      flashData <= rxFourNibbles;
298       --      --indicate write operation to FSM
299       --      we_2fsm <= '1';
300       --    end if;
301       --  end if;
302       --end process setOutputs;
303
304       --set startAddr register for FSM (24 bit signal)
305       setStartAddr: process (clk, rst)
306       begin
307         if rst = '1' then
308           startAddr <= (others => '0');
309         elsif rising_edge(clk) then
310           if counterOF6 = '1' then
311             if startAddrCommand = '1' then
312               --forward start address to FSM
313               startAddr <= rxSixNibbles;
314             end if;
315           end if;
316         end if;
317       end process setStartAddr;
318
319       --setStartAddr: process (counterOF6, startAddrCommand,
    rxSixNibbles)
320       --begin
321       --  if counterOF6 = '1' then
322       --    if startAddrCommand = '1' then
323       --      --forward start address to FSM
324       --      startAddr <= rxSixNibbles;
325       --    end if;
326       --  end if;
327       --end process setStartAddr;
328
329
330
331       --set erase signals for FSM
332       setEraseSignals: process (eraseCommand)
333       begin
334         if eraseCommand = '1' then
335            erase_2fsm <= '1';
336         else
337            erase_2fsm <= '0';
338         end if;
339       end process setEraseSignals;
340
341       --set read signals for FSM
```

```
342         setReadSignals: process (readCommand)
343         begin
344           if readCommand = '1' then
345              oe_2fsm <= '1';
346           else
347              oe_2fsm <= '0';
348           end if;
349         end process setReadSignals;
350
351
352
353         ----test operation
354         --
355         --test: process(rst, clk)
356         --begin
357         --  if rst = '1' then
358         --      output <= '0';
359         --  elsif rising_edge(clk) then
360         --      if rxData1 = "01100010" then
361         --         output <= '1';
362         --      end if;
363         --  end if;
364         --end process test;
365
366
367
368         --checkData: process (clk, rst)
369         --  variable CBarrived: std_uLogic;
370         --begin
371         --  if rst = '1' then
372         --      CBarrived := '0';
373         --      erase_2fsm <= '0';
374         --      ce_2fsm <= '0';
375         --      oe_2fsm <= '0';
376         --      we_2fsm <= '0';
377         --  elsif rising_edge(clk) then
378         --      if rxRd1 = '1' then
379         --        --new databyte arrived
380         --
381         --        if CBarrived = '1' then
382         --          --previous byte was CONTROLBYTE (CB)
383         --          if rxData1(rxData1'high downto rxData1'high-2) =
    HEADER101 then
384         --            --current byte is starting with 101 sequence
385         --            --set signals
386         --            erase_2fsm <= rxData1(ERASEBIT);
387         --            ce_2fsm <= rxData1(CEBIT);
388         --            oe_2fsm <= rxData1(OEBIT);
389         --            we_2fsm <= rxData1(WEBIT);
390         --            CBarrived := '0';
391         --           end if;
392         --        end if;
393         --
394         --        if rxData1 = CB then
395         --          --byte is a CONTROLBYTE (CB)
396         --          CBarrived := '1';
397         --          erase_2fsm <= '0';
398         --          ce_2fsm <= '0';
399         --          oe_2fsm <= '0';
400         --          we_2fsm <= '0';
```

```
401         --        end if;
402         --
403         --      end if;
404         --  end if;
405         --end process checkData;
406
407
408
409
410         --if rxData1(rxData1'high downto rxData1'high-2) = "101" then
411         --current byte is an OUTPUTBYTE containing CS, OE, WE info
412
413         --ce_2fsm <= rxData1(CEBIT);
414         --oe_2fsm <= rxData1(OEBIT);
415         --we_2fsm <= rxData1(WEBIT);
416
417
418
419      -- Instance port mappings.
420
421   END struct;
```

Beilage 19

FLASH Perl Scripts

```
 1 #includes
 2 #====================================================================
 3 use Win32::SerialPort;
 4 use Term::ReadKey;
 5 use Switch;
 6 #====================================================================
 7
 8 #constants/defines
 9 #====================================================================
10 my $SRECORD_DATA_LEN = 32;      #data length in sRecord
11 my $PortName = 'COM1';          #port for serial interface
12 my $ERASE_COMMAND = 'x';        #Erase Command
13 my $START_ADDR_COMMAND = 's';   #Start Address Command
14 my $WRITE_COMMAND = 'w';        #Write Command
15 my $LENGTH_COMMAND = 'l';       #Length Command
16 my $READ_COMMAND = 'r';         #Read Command
17 my $ERASE_ACK = 'o';            #Erase ACK
18 my $ERASE_NACK = 'n';           #Erase NACK
19 my $WRITE_ACK = 'v';            #Write ACK
20 my $WRITE_NACK = 'm';           #Write NACK
21 my $READ_ACK = 'p';             #READ ACK
22 my $READ_NACK = 'i';            #READ NACK
23 my $START_ADDR = '00000';      #Start Address
24
25 #====================================================================
26
27 # objects
28 #====================================================================
29 # create new serial port
30 $PortObj = new Win32::SerialPort ($PortName) || die ;
31 #====================================================================
32
33 # variables
34 #====================================================================
35
36 #for serial port usage
37 my $serialPortReadBuffer = 4096;
38 my $serialPortWriteBuffer = 4096;
39 my $baud;
40 my $parity;
41 my $data;
42 my $stop;
43 my $hshake;
44
45 #for input handlinig
46 my $gotit = "";
47 my $match1 = "";
48 my $gotitSerial = "";
49
50 #for memory file handling
51 my $sRecordType;
52 my $addressLength;
53 my $sRecordLength;
54 my $currentAddress;
55 my $currentData;
56 my $currentByte;
57 my $dataLenght;
58
59 # subroutines
60 #====================================================================
```

```
 61 sub ascii_to_hex ($)
 62 {
 63     ## Convert each ASCII character to a two-digit hex number.
 64     (my $str = shift) =~ s/(.|\n)/sprintf("%02lx", ord $1)/eg;
 65     return $str;
 66 }
 67
 68 sub bin2dec
 69 {
 70     return unpack("N", pack("B32", substr("0" x 32 . shift, -32)));
 71 }
 72
 73 #=====================================================================
 74
 75 #start of program
 76 #=====================================================================
 77
 78 #init serial interface
 79 #=======================
 80 $PortObj->baudrate(9600);
 81 $PortObj->parity("none");
 82 $PortObj->databits(8);
 83 $PortObj->stopbits(1);
 84 $PortObj->handshake("none");
 85 $PortObj-> write_settings;
 86
 87 $PortObj->buffers($serialPortReadBuffer, $serialPortWriteBuffer);
 88 $PortObj->are_match("\r");    # wait for enter input
 89 $PortObj->lookclear;          # empty buffers
 90
 91 $baud = $PortObj->baudrate;
 92
 93 #display welcome screen
 94 #=======================
 95 #print "\n\rbaud rate = $baud\n\r";
 96 print("\n\rStarting FLASH ERASE App...\n\r");
 97
 98
 99 #erase flash
100 #=======================
101
102 ##send erase command
103 print "\n\rERASING NOW...!\n\r";
104 $PortObj->write($ERASE_COMMAND);
105 #
106 #wait for erase ack
107 do{
108     $gotitSerial = "";                      #clear input
109
110     until ("" ne $gotitSerial)
111     {
112         $gotitSerial = $PortObj->lookfor;   # poll until data ready
113         last if ($gotitSerial);
114         $match1 = $PortObj->matchclear;     # match is first thing
    received
115         last if ($match1);
116         sleep 1;                            # polling sample time
117     }
118     #print "\n\r$gotitSerial";
119
```

```
120 }until($gotitSerial eq $ERASE_ACK || $ERASE_NACK);
121
122 if ($gotitSerial eq $ERASE_NACK)
123 {
124     #display erase error message
125     print "\n\rERASE ERROR!\n\r";
126 }
127 else
128 {
129     #display erase confirm message
130     print "\n\rERASE COMPLETED!\n\r";
131 }
```

```
 1 #includes
 2 #====================================================================
 3 use Win32::SerialPort;
 4 use Term::ReadKey;
 5 use Switch;
 6 #====================================================================
 7
 8 #constants/defines
 9 #====================================================================
10 my $SRECORD_DATA_LEN = 32;      #data length in sRecord
11 my $START_ADDR_LEN = 6;         #start addr length to send
12 my $PortName = 'COM1';          #port for serial interface
13 my $ERASE_COMMAND = 'x';        #Erase Command
14 my $START_ADDR_COMMAND = 's';   #Start Address Command
15 my $WRITE_COMMAND = 'w';        #Write Command
16 my $LENGTH_COMMAND = 'l';       #Length Command
17 my $READ_COMMAND = 'r';         #Read Command
18 my $ERASE_ACK = 'o';            #Erase ACK
19 my $ERASE_NACK = 'n';           #Erase NACK
20 my $WRITE_ACK = 'v';            #Write ACK
21 my $WRITE_NACK = 'm';           #Write NACK
22 my $READ_ACK = 'p';             #READ ACK
23 my $READ_NACK = 'i';            #READ NACK
24 my $START_ADDR = '00000';      #Start Address
25
26 #====================================================================
27
28 # objects
29 #====================================================================
30 # create new serial port
31 $PortObj = new Win32::SerialPort ($PortName) || die ;
32 #====================================================================
33
34 # variables
35 #====================================================================
36
37 #for serial port usage
38 my $serialPortReadBuffer = 4096;
39 my $serialPortWriteBuffer = 4096;
40 my $baud;
41 my $parity;
42 my $data;
43 my $stop;
44 my $hshake;
45
46 #for input handlinig
47 my $gotit = "";
48 my $match1 = "";
49 my $gotitSerial = "";
50
51 #for memory file handling
52 my $sRecordType;
53 my $addressLength;
54 my $sRecordLength;
55 my $currentAddress;
56 my $currentAddressSend;
57 my $currentAddressMSB;
58 my $currentData;
59 my $currentByte;
60 my $dataLenght;
```

```
 61
 62 # subroutines
 63 #====================================================================
 64 sub ascii_to_hex ($)
 65 {
 66     ## Convert each ASCII character to a two-digit hex number.
 67     (my $str = shift) =~ s/(.|\n)/sprintf("%02lx", ord $1)/eg;
 68     return $str;
 69 }
 70
 71 sub bin2dec
 72 {
 73     return unpack("N", pack("B32", substr("0" x 32 . shift, -32)));
 74 }
 75
 76 #====================================================================
 77
 78 #start of program
 79 #====================================================================
 80
 81 #init serial interface
 82 #========================
 83 $PortObj->baudrate(9600);
 84 $PortObj->parity("none");
 85 $PortObj->databits(8);
 86 $PortObj->stopbits(1);
 87 $PortObj->handshake("none");
 88 $PortObj-> write_settings;
 89
 90 $PortObj->buffers($serialPortReadBuffer, $serialPortWriteBuffer);
 91 $PortObj->are_match("\r");    # wait for enter input
 92 $PortObj->lookclear;          # empty buffers
 93
 94 $baud = $PortObj->baudrate;
 95
 96 #display welcome screen
 97 #========================
 98 #print "\n\rbaud rate = $baud\n\r";
 99 print("\n\rStarting FLASH WRITE App...\n\r");
100
101 #ask for and open file
102 #========================
103
104 print("\n\rPlease enter the .srec file name including file extension:
    \n\r");
105 $gotit = <>;
106 #$gotit = "flashtest.srec";
107
108 print "\n\rFilename = $gotit\n\r";
109
110 #open the .srec file
111 open(FILE, $gotit) || die("Could not open file!");
112
113 #reading the file and save it to raw_data
114 @raw_data=<FILE>;
115
116 #closing the file
117 close(FILE);
118
119 #write flash
```

```
120 #========================
121
122 print "\n\rWRITING DATA NOW...!\n\r\n\r\n\r";
123
124     #clear input
125     $gotitSerial = "";
126
127     #process the data
128     foreach $sRecordLine (@raw_data)
129     {
130         #remove \n character
131         #print "\n\rcurrent record = $sRecordLine\n\r";
132
133         #get record type, S1, S3...
134         $sRecordType = substr($sRecordLine, 1, 1);
135
136         #get record length
137         $sRecordLength = substr($sRecordLine, 2, 2);
138         #print "\n\rsRecord Length = $sRecordLength\n\r";
139
140         switch ($sRecordType)
141         {
142
143             case "0"
144             {
145                 #print "\n\rsRecord Type = $sRecordType\n\r";
146                 #print "S0 Record found\n\r";
147             }
148             case "1"
149             {
150
151                 #clear input
152                 $gotitSerial = "";
153
154                 #print "\n\rsRecord Type = $sRecordType\n\r";
155
156                 #send start address command
157                 $PortObj->write($START_ADDR_COMMAND);
158
159                 #get address from record
160                 $addressLength = 4;
161                 $currentAddress = substr($sRecordLine, 4,
    $addressLength);
162                 #print "\n\rCurrent Address = $currentAddress\n\r";
163
164                 #check for flash address and send only flash
    addresses
165                 $currentAddressMSB = substr($currentAddress, 0, 1);
166                 #print "\n\r\n\rAddress Temp:
    $currentAddressMSB\n\r";
167                 if ($currentAddressMSB == 0)
168                 {
169
170                     #send start address
171                     #print "\n\rAddress $currentAddress\n\r";
172                     $currentAddressSend = sprintf("%06s",
    $currentAddress);
173                     print "\n\rAddress $currentAddressSend\n\r";
174                     $PortObj->write($currentAddressSend);
175
```

```
176                     #send length command
177                     $PortObj->write($LENGTH_COMMAND);
178
179                     #send length
180                     #print "\n\rDataLength $SRECORD_DATA_LEN\n\r";
181                     $dataLenght = sprintf("%04x", $SRECORD_DATA_LEN);
182                     #print "\n\rDataLength $dataLenght\n\r";
183                     $PortObj->write($dataLenght);
184
185                     #send write command
186                     $PortObj->write($WRITE_COMMAND);
187
188                     #get data from record
189                     $currentData = substr($sRecordLine,
    4+$addressLength, $SRECORD_DATA_LEN);
190                     print "Current sRecord = $currentData";
191
192                     #send data, byte per byte over the serial
    interface
193                     for($i = 0; $i <= 3; $i++)
194                     {
195                         $currentByte = substr($currentData, $i*8, 8);
196                         print "\n\rWriting Data: $currentByte";
197                         $PortObj->write($currentByte);
198                     }
199
200
201                     #wait for write ack
202                     do{
203
204                     until ("" ne $gotitSerial)
205                         {
206                             $gotitSerial = $PortObj->lookfor;   #
    poll until data ready
207                             last if ($gotitSerial);
208                             $match1 = $PortObj->matchclear;     #
    match is first thing received
209                             last if ($match1);
210                             sleep 1;                            #
    polling sample time
211                         }
212
213                     }until($gotitSerial eq $WRITE_ACK ||
    $WRITE_NACK);
214                     if ($gotitSerial eq $WRITE_NACK)
215                     {
216                         #display write error message
217                         print "\n\r\n\rWRITE ERROR!\n\r";
218                         exit;
219                     }
220                 }
221                 else
222                 {
223                     #display write confirm message
224                     print "\n\rWRITE COMPLETED!\n\r";
225                     exit;
226                 }
227
228             }
229             case "2"
```

```
230             {
231
232                 #clear input
233                 $gotitSerial = "";
234
235                 #print "\n\rsRecord Type = $sRecordType\n\r";
236
237                 #send start address command
238                 $PortObj->write($START_ADDR_COMMAND);
239
240                 #get address from record
241                 $addressLength = 6;
242                 $currentAddress = substr($sRecordLine, 4,
    $addressLength);
243                 #print "\n\rCurrent Address = $currentAddress\n\r";
244
245                 #check for flash address and send only flash
    addresses
246                 $currentAddressMSB = substr($currentAddress, 0, 1);
247                 #print "\n\r\n\rAddress Temp:
    $currentAddressMSB\n\r";
248                 if ($currentAddressMSB == 0)
249                 {
250
251                     #send start address
252                     #print "\n\rAddress $currentAddress\n\r";
253                     $currentAddressSend = $currentAddress;
254                     print "\n\rAddress $currentAddressSend\n\r";
255                     $PortObj->write($currentAddressSend);
256
257                     #send length command
258                     $PortObj->write($LENGTH_COMMAND);
259
260                     #send length
261                     #print "\n\rDataLength $SRECORD_DATA_LEN\n\r";
262                     $dataLenght = sprintf("%04x", $SRECORD_DATA_LEN);
263                     #print "\n\rDataLength $dataLenght\n\r";
264                     $PortObj->write($dataLenght);
265
266                     #send write command
267                     $PortObj->write($WRITE_COMMAND);
268
269                     #get data from record
270                     $currentData = substr($sRecordLine,
    4+$addressLength, $SRECORD_DATA_LEN);
271                     print "Current sRecord = $currentData";
272
273                     #send data, byte per byte over the serial
    interface
274                     for($i = 0; $i <= 3; $i++)
275                     {
276                         $currentByte = substr($currentData, $i*8, 8);
277                         print "\n\rWriting Data: $currentByte";
278                         $PortObj->write($currentByte);
279                     }
280
281
282                     #wait for write ack
283                     do{
284
```

```
285                         until ("" ne $gotitSerial)
286                         {
287                             $gotitSerial = $PortObj->lookfor;   # poll until data ready
288                             last if ($gotitSerial);
289                             $match1 = $PortObj->matchclear;     # match is first thing received
290                             last if ($match1);
291                             sleep 1;                            # polling sample time
292                         }
293
294                     }until($gotitSerial eq $WRITE_ACK || $WRITE_NACK);
295                     if ($gotitSerial eq $WRITE_NACK)
296                     {
297                         #display write error message
298                         print "\n\r\n\rWRITE ERROR!\n\r";
299                         exit;
300                     }
301                 }
302                 else
303                 {
304                     #display write confirm message
305                     print "\n\rWRITE COMPLETED!\n\r";
306                     exit;
307                 }
308
309             }
310             case "3"
311             {
312
313                 #clear input
314                 $gotitSerial = "";
315
316                 #print "\n\rsRecord Type = $sRecordType\n\r";
317
318                 #send start address command
319                 $PortObj->write($START_ADDR_COMMAND);
320
321                 #get address from record
322                 $addressLength = 8;
323                 $currentAddress = substr($sRecordLine, 4, $addressLength);
324                 #print "\n\rCurrent Address = $currentAddress\n\r";
325
326                 #check for flash address and send only flash addresses
327                 $currentAddressMSB = substr($currentAddress, 0, 1);
328                 #print "\n\r\n\rAddress Temp: $currentAddressMSB\n\r";
329                 if ($currentAddressMSB == 0)
330                 {
331
332                     #send start address
333                     #print "\n\rAddress $currentAddress\n\r";
334                     $currentAddressSend = substr($currentAddress, 2, $START_ADDR_LEN);
335                     print "\n\rAddress $currentAddressSend\n\r";
336                     $PortObj->write($currentAddressSend);
```

```
337
338                     #send length command
339                     $PortObj->write($LENGTH_COMMAND);
340
341                     #send length
342                     #print "\n\rDataLength $SRECORD_DATA_LEN\n\r";
343                     $dataLenght = sprintf("%04x", $SRECORD_DATA_LEN);
344                     #print "\n\rDataLength $dataLenght\n\r";
345                     $PortObj->write($dataLenght);
346
347                     #send write command
348                     $PortObj->write($WRITE_COMMAND);
349
350                     #get data from record
351                     $currentData = substr($sRecordLine,
    4+$addressLength, $SRECORD_DATA_LEN);
352                     print "Current sRecord = $currentData";
353
354                     #send data, byte per byte over the serial
    interface
355                     for($i = 0; $i <= 3; $i++)
356                     {
357                         $currentByte = substr($currentData, $i*8, 8);
358                         print "\n\rWriting Data: $currentByte";
359                         $PortObj->write($currentByte);
360                     }
361
362
363                     #wait for write ack
364                     do{
365
366                         until ("" ne $gotitSerial)
367                         {
368                             $gotitSerial = $PortObj->lookfor;   #
    poll until data ready
369                             last if ($gotitSerial);
370                             $match1 = $PortObj->matchclear;     #
    match is first thing received
371                             last if ($match1);
372                             sleep 1;                            #
    polling sample time
373                         }
374
375                     }until($gotitSerial eq $WRITE_ACK ||
    $WRITE_NACK);
376                     if ($gotitSerial eq $WRITE_NACK)
377                     {
378                         #display write error message
379                         print "\n\r\n\rWRITE ERROR!\n\r";
380                         exit;
381                     }
382                 }
383                 else
384                 {
385                     #display write confirm message
386                     print "\n\rWRITE COMPLETED!\n\r";
387                     exit;
388                 }
389
390             }
```

```
391             case "7"
392             {
393                 #print "\n\rsRecord Type = $sRecordType\n\r";
394                 #print "S7 Record found\n\r";
395             }
396             else
397             {
398                 print "\n\rsRecord unknown\n\r";
399             }
400         }
401     }
402
```

```
 1 #includes
 2 #====================================================================
 3 use Win32::SerialPort;
 4 use Term::ReadKey;
 5 use Switch;
 6 #====================================================================
 7
 8 #constants/defines
 9 #====================================================================
10 my $SRECORD_DATA_LEN = 32;      #data length in sRecord
11 my $PortName = 'COM1';          #port for serial interface
12 my $ERASE_COMMAND = 'x';        #Erase Command
13 my $START_ADDR_COMMAND = 's';   #Start Address Command
14 my $WRITE_COMMAND = 'w';        #Write Command
15 my $LENGTH_COMMAND = 'l';       #Length Command
16 my $READ_COMMAND = 'r';         #Read Command
17 my $ERASE_ACK = 'o';            #Erase ACK
18 my $ERASE_NACK = 'n';           #Erase NACK
19 my $WRITE_ACK = 'v';            #Write ACK
20 my $WRITE_NACK = 'm';           #Write NACK
21 my $READ_ACK = 'p';             #READ ACK
22 my $READ_NACK = 'i';            #READ NACK
23 my $START_ADDR = '00000';       #Start Address
24 my $targetFile = "flashReadout.srec";   #target file for output
25
26 #====================================================================
27
28 # objects
29 #====================================================================
30 # create new serial port
31 $PortObj = new Win32::SerialPort ($PortName) || die ;
32 #====================================================================
33
34 # variables
35 #====================================================================
36
37 #for serial port usage
38 my $serialPortReadBuffer = 4096;
39 my $serialPortWriteBuffer = 4096;
40 my $baud;
41 my $parity;
42 my $data;
43 my $stop;
44 my $hshake;
45
46 #for input handlinig
47 my $gotit = "";
48 my $match1 = "";
49 my $gotitSerial = "";
50
51 #for memory file handling
52 my $sRecordType;
53 my $addressLength;
54 my $sRecordLength;
55 my $currentAddress;
56 my $currentAddresstemp;
57 my $currentData;
58 my $currentByte;
59 my $dataLenght;
60 my $readCounter;
```

```
 61
 62 # subroutines
 63 #====================================================================
 64 sub ascii_to_hex ($)
 65 {
 66     ## Convert each ASCII character to a two-digit hex number.
 67     (my $str = shift) =~ s/(.|\n)/sprintf("%02lx", ord $1)/eg;
 68     return $str;
 69 }
 70
 71 sub bin2dec
 72 {
 73     return unpack("N", pack("B32", substr("0" x 32 . shift, -32)));
 74 }
 75
 76 #====================================================================
 77
 78 #start of program
 79 #====================================================================
 80
 81 #init serial interface
 82 #========================
 83 $PortObj->baudrate(9600);
 84 $PortObj->parity("none");
 85 $PortObj->databits(8);
 86 $PortObj->stopbits(1);
 87 $PortObj->handshake("none");
 88 $PortObj-> write_settings;
 89
 90 $PortObj->buffers($serialPortReadBuffer, $serialPortWriteBuffer);
 91 $PortObj->are_match("\r");    # wait for enter input
 92 $PortObj->lookclear;          # empty buffers
 93
 94 $baud = $PortObj->baudrate;
 95
 96 #display welcome screen
 97 #========================
 98 #print "\n\rbaud rate = $baud\n\r";
 99 print("\n\rStarting FLASH READ App...\n\r");
100
101
102 #read flash
103 #========================
104
105 print "\n\rPlease enter the START ADDRESS:\n\r";
106 $gotit = <>;
107
108 $currentAddress = $gotit;
109 $StartAddrTemp = sprintf("%06x", $gotit);
110
111 #send start address command
112 $PortObj->write($START_ADDR_COMMAND);
113
114 #send start address
115 $PortObj->write($StartAddrTemp);
116
117 print "\n\rPlease enter the amount of SRECs to read:\n\r";
118 $gotit = <>;
119 $dataLenghtTemp = sprintf("%04x", $gotit*32);
120
```

```
  121 #send length command
  122 $PortObj->write($LENGTH_COMMAND);
  123
  124 $PortObj->write($dataLenghtTemp);
  125
  126 #send read command
  127 $PortObj->write($READ_COMMAND);
  128
  129
  130 #start reading and write to file
  131 print "\n\rREADING DATA NOW AND WRITING to flashReadout.srec!\n\r";
  132
  133 #open target file and set permissions
  134 open(FILE,">$targetFile") || die("Cannot Open File");
  135 flock(FILE, LOCK_EX);
  136 seek(FILE, 0, SEEK_SET);
  137
  138 #add fake S0 record
  139 print FILE "S0XXFollowing Data read out from FLASH\n";
  140
  141 #wait for read ack and write incoming data from Flash to target file
  142 do{
  143
  144     if($readCounter == 0)
  145     {
  146         #print S-Record Header plus address with fixed incrementation
  147         $currentAddresstemp = sprintf("%08x", $currentAddress);
  148         #print "\n\rAddress $currentAddresstemp\n\r";
  149         print FILE "S315$currentAddresstemp";
  150         $currentAddress = $currentAddress + 16;
  151     }
  152
  153     $gotitSerial = "";                      #clear input
  154
  155     until ("" ne $gotitSerial)
  156     {
  157         $gotitSerial = $PortObj->lookfor;   # poll until data ready
  158         last if ($gotitSerial);
  159         $match1 = $PortObj->matchclear;     # match is first thing
      received
  160         last if ($match1);
  161         sleep 1;                            # polling sample time
  162
  163     }
  164     $readCounter++;
  165     #print "\n\r$gotitSerial";
  166     if($gotitSerial ne $READ_ACK)
  167     {
  168         #write input to file
  169         #print "\n\r$gotitSerial";
  170         print FILE "$gotitSerial";
  171     }
  172     if($gotitSerial eq $READ_ACK)
  173     {
  174         close(FILE);
  175
  176         #read old file
  177         open(FILE, $targetFile) || die("Cannot Open File");
  178         @raw_data=<FILE>;
  179         close(FILE);
```

```
180
181         #remove last line
182         splice(@raw_data,$#raw_data,1);
183
184         #write new file
185         open(FILE,">$targetFile") || die("Cannot Open File");
186         flock(FILE, LOCK_EX);
187         seek(FILE, 0, SEEK_SET);
188         print FILE @raw_data;
189         close(FILE);
190     }
191
192     if ($readCounter >= $SRECORD_DATA_LEN)
193     {
194         #end of line found
195         print FILE "XX\n";
196         $readCounter = 0;
197     }
198
199 }until($gotitSerial eq $READ_ACK);
200
201 #close target file
202 close(FILE);
203
204 if ($gotitSerial eq $READ_NACK)
205 {
206     print "\n\rREAD ERROR!\n\r";
207 }
208 else
209 {
210     print "\n\rREADING DONE...!\n\r";
211 }
212
```

Beilage 20

FLASH Programmation .ucf (I/O Mapping)

```
 1  #PACE: Start of Constraints generated by PACE
 2
 3  #PACE: Start of PACE I/O Pin Assignments
 4  NET "addr<0>"  LOC = "A11"  ;
 5  NET "addr<10>"  LOC = "K13"  ;
 6  NET "addr<11>"  LOC = "L15"  ;
 7  NET "addr<12>"  LOC = "L16"  ;
 8  NET "addr<13>"  LOC = "T18"  ;
 9  NET "addr<14>"  LOC = "R18"  ;
10  NET "addr<15>"  LOC = "T17"  ;
11  NET "addr<16>"  LOC = "U18"  ;
12  NET "addr<17>"  LOC = "T16"  ;
13  NET "addr<18>"  LOC = "U15"  ;
14  NET "addr<19>"  LOC = "V15"  ;
15  NET "addr<1>"  LOC = "J13"  ;
16  NET "addr<20>"  LOC = "R14"  ;
17  NET "addr<21>"  LOC = "T14"  ;
18  NET "addr<22>"  LOC = "R13"  ;
19  NET "addr<23>"  LOC = "P13"  ;
20  NET "addr<2>"  LOC = "J12"  ;
21  NET "addr<3>"  LOC = "J14"  ;
22  NET "addr<4>"  LOC = "J15"  ;
23  NET "addr<5>"  LOC = "J16"  ;
24  NET "addr<6>"  LOC = "J17"  ;
25  NET "addr<7>"  LOC = "K14"  ;
26  NET "addr<8>"  LOC = "K15"  ;
27  NET "addr<9>"  LOC = "K12"  ;
28  NET "cen"  LOC = "M16"  ;
29  NET "clk"  LOC = "A10"  ;
30  NET "data<0>"  LOC = "H17"  ;
31  NET "data<10>"  LOC = "C18"  ;
32  NET "data<11>"  LOC = "G14"  ;
33  NET "data<12>"  LOC = "F14"  ;
34  NET "data<13>"  LOC = "D11"  ;
35  NET "data<14>"  LOC = "F15"  ;
36  NET "data<15>"  LOC = "D16"  ;
37  NET "data<1>"  LOC = "F17"  ;
38  NET "data<2>"  LOC = "D17"  ;
39  NET "data<3>"  LOC = "C17"  ;
40  NET "data<4>"  LOC = "G15"  ;
41  NET "data<5>"  LOC = "E11"  ;
42  NET "data<6>"  LOC = "G16"  ;
43  NET "data<7>"  LOC = "E13"  ;
44  NET "data<8>"  LOC = "F18"  ;
45  NET "data<9>"  LOC = "E17"  ;
46  NET "flashResetn"  LOC = "M18"  ;
47  NET "led1"  LOC = "B16"  ;
48  NET "led2"  LOC = "A16"  ;
49  NET "mezzaIRQ0"  LOC = "G13"  ;
50  NET "mezzaIRQ1"  LOC = "F12"  ;
51  NET "mezzaIRQ2"  LOC = "E12"  ;
52  NET "oen"  LOC = "H15"  ;
53  NET "rstn"  LOC = "A15" | PULLUP ;
54  NET "rxd0"  LOC = "V2"  ;
55  NET "rxd1"  LOC = "U1"  ;
56  NET "sts"  LOC = "C11" | PULLUP ;
57  NET "txd0"  LOC = "T1"  ;
58  NET "txd1"  LOC = "P1"  ;
59  NET "wen"  LOC = "H16"  ;
60
```

```
   61  #PACE: Start of PACE Area Constraints
   62
   63  #PACE: Start of PACE Prohibit Constraints
   64
   65  #PACE: End of Constraints generated by PACE
```

Beilage 21

LEON Testapplikation readtest.c

```
 1 /*
 2  * readtest.c
 3  *
 4  *  Created on: 24 juin 2009
 5  *      Author: Thomas
 6  */
 7
 8 //defines
 9 #define RAMAREA (0x40000000)
10 #define MEMCTRLAREA (0x80000200)
11
12 //main loop
13 int main(int argc, char *argv[])
14 {
15   //define pointers to memory control area
16   volatile unsigned int *memctrl = (volatile unsigned int
   *)MEMCTRLAREA;
17
18   //define pointers to ram area
19   volatile unsigned char *ram_char=(volatile unsigned char *)RAMAREA;
20   volatile unsigned int *ram_int=(volatile unsigned int *)RAMAREA;
21
22
23   //set MCTRL registers
24   memctrl[0] = 0x1000010F;    //MCFG1 no PROM WRITE, 16 Bit mode
25   memctrl[1] = 0x9030605A;    //CAS = 2, no SRAM, SDRAM config...
26   memctrl[2] = 0x00400000;  // MCFG3, set sdram refresh 1024 =
   10000000000b
27
28   //variables
29   int i = 0;
30   int last = 0;
31
32   //write SDRAM
33   ram_int[10] = 50;
34   //read SDRAM
35   last = ram_int[10];
36
37   //read check
38   if(last == 50)
39   {
40     //correct readout, continue with more r/w operations
41     ram_int[100] = 200;
42
43     for (i = 200; i<205; i++)
44     {
45       ram_int[i] = i;
46     }
47
48     for (i = 200; i<205; i++)
49     {
50       last = ram_int[i];
51     }
52   }
53   else
54   {
55     //wrong readout, write a 32 bit value
56     ram_int[150] = 0x12345678;
57   }
58
```

```
   59    while(1)
   60    {
   61      //loop
   62    }
   63
   64    return 0;
   65  }
   66
```

Beilage 22

VHDL Code FLASH Tester

```
 1 --
 2 -- VHDL Architecture
   Flash_Prog_test.Flash_Prog_tester.Flash_Prob_tester
 3 --
 4 -- Created:
 5 --          by - Thomas.UNKNOWN (WE2778)
 6 --          at - 08:39:27 27.05.2009
 7 --
 8 -- using Mentor Graphics HDL Designer(TM) 2007.1a (Build 13)
 9 --
10 ARCHITECTURE Flash_Prob_tester OF Flash_Prog_tester IS
11
12   constant clock66MFrequency: real := 66.0E6;
13   constant clock66MPeriod: time := (1.0/clock66MFrequency) * 1 sec;
14   signal clock66M_int: std_uLogic := '1';
15
16   constant rs232Frequency: real := 9600.0;
17   constant rs232Period: time := (1.0/rs232Frequency) * 1 sec;
18   signal rs232_0Send: std_uLogic;
19   signal rs232_0Data: character;
20   signal rs232_1Send: std_uLogic;
21   signal rs232_1Data: character;
22   --signal rs232_1Data: unsigned(7 downto 0);
23
24 BEGIN
25
26 -------------------------------------------------------
27 -- reset and clocks
28   rstn <= '0', '1' after 10*clock66MPeriod;
29   --RP_n <= '0', '1' after 10*clock66MPeriod;
30
31   clock66M_int <= not clock66M_int after clock66MPeriod/2;
32   clk <= transport clock66M_int after clock66MPeriod*9/10;
33
34 --  sts <= '1';
35 --  lb <= '0';
36 --  ub <= '0';
37
38   BYTE_n <= '1'; --16 bit mode
39
40 -------------------------------------------------------
41 -- RS232 bridge
42   process
43   begin
44
45     rs232_0Send <= '0';
46     rs232_1Send <= '0';
47     wait for 4*rs232Period;
48
49     --send CR
50     rs232_0Data <= cr;
51     rs232_0Send <= '1', '0' after 1 ns;
52     wait for 10*rs232Period;
53
54     --send ERASE
55     rs232_0Data <= 'x'; --erase!
56     rs232_0Send <= '1', '0' after 1 ns;
57     wait for 10*rs232Period;
58
59     --if txDataOut = "11001010" then
```

```
 60
 61 ------------------------------------------------------
 62
 63       --send START ADDR
 64       rs232_0Data <= 's';
 65       rs232_0Send <= '1', '0' after 1 ns;
 66       wait for 10*rs232Period;
 67
 68       --send DATABYTE
 69       rs232_0Data <= '0';
 70       rs232_0Send <= '1', '0' after 1 ns;
 71       wait for 10*rs232Period;
 72
 73       --send DATABYTE
 74       rs232_0Data <= '0';
 75       rs232_0Send <= '1', '0' after 1 ns;
 76       wait for 10*rs232Period;
 77
 78       --send DATABYTE
 79       rs232_0Data <= '0';
 80       rs232_0Send <= '1', '0' after 1 ns;
 81       wait for 10*rs232Period;
 82
 83       --send DATABYTE
 84       rs232_0Data <= '0';
 85       rs232_0Send <= '1', '0' after 1 ns;
 86       wait for 10*rs232Period;
 87
 88       --send DATABYTE
 89       rs232_0Data <= '0';
 90       rs232_0Send <= '1', '0' after 1 ns;
 91       wait for 10*rs232Period;
 92
 93       --send DATABYTE
 94       rs232_0Data <= '0';
 95       rs232_0Send <= '1', '0' after 1 ns;
 96       wait for 10*rs232Period;
 97 ------------------------------------------------------
 98
 99       --send LENGTH
100       rs232_0Data <= 'l'; --lenght!
101       rs232_0Send <= '1', '0' after 1 ns;
102       wait for 10*rs232Period;
103
104       --send DATABYTE
105       rs232_0Data <= '0';
106       rs232_0Send <= '1', '0' after 1 ns;
107       wait for 10*rs232Period;
108
109       --send DATABYTE
110       rs232_0Data <= '0';
111       rs232_0Send <= '1', '0' after 1 ns;
112       wait for 10*rs232Period;
113
114       --send DATABYTE
115       rs232_0Data <= '0';
116       rs232_0Send <= '1', '0' after 1 ns;
117       wait for 10*rs232Period;
118
119       --send DATABYTE
```

```
120         rs232_0Data <= '8';
121         rs232_0Send <= '1', '0' after 1 ns;
122         wait for 10*rs232Period;
123 
124  ------------------------------------------------------
125 
126         --send WRITE
127         rs232_0Data <= 'w'; --write!
128         rs232_0Send <= '1', '0' after 1 ns;
129         wait for 10*rs232Period;
130 
131         --send DATABYTE
132         rs232_0Data <= '1';
133         rs232_0Send <= '1', '0' after 1 ns;
134         wait for 10*rs232Period;
135 
136         --send DATABYTE
137         rs232_0Data <= '2';
138         rs232_0Send <= '1', '0' after 1 ns;
139         wait for 10*rs232Period;
140 
141         --send DATABYTE
142         rs232_0Data <= '3';
143         rs232_0Send <= '1', '0' after 1 ns;
144         wait for 10*rs232Period;
145 
146         --send DATABYTE
147         rs232_0Data <= '4';
148         rs232_0Send <= '1', '0' after 1 ns;
149         wait for 10*rs232Period;
150 
151         --send DATABYTE
152         rs232_0Data <= 'A';
153         rs232_0Send <= '1', '0' after 1 ns;
154         wait for 10*rs232Period;
155 
156         --send DATABYTE
157         rs232_0Data <= 'B';
158         rs232_0Send <= '1', '0' after 1 ns;
159         wait for 10*rs232Period;
160 
161         --send DATABYTE
162         rs232_0Data <= 'C';
163         rs232_0Send <= '1', '0' after 1 ns;
164         wait for 10*rs232Period;
165 
166         --send DATABYTE
167         rs232_0Data <= 'D';
168         rs232_0Send <= '1', '0' after 1 ns;
169         wait for 10*rs232Period;
170 
171         --send DATABYTE
172         rs232_0Data <= '1';
173         rs232_0Send <= '1', '0' after 1 ns;
174         wait for 10*rs232Period;
175 
176         --send DATABYTE
177         rs232_0Data <= '2';
178         rs232_0Send <= '1', '0' after 1 ns;
179         wait for 10*rs232Period;
```

```
180
181         --send DATABYTE
182         rs232_0Data <= '3';
183         rs232_0Send <= '1', '0' after 1 ns;
184         wait for 10*rs232Period;
185
186         --send DATABYTE
187         rs232_0Data <= '4';
188         rs232_0Send <= '1', '0' after 1 ns;
189         wait for 10*rs232Period;
190
191  ------------------------------------------------------
192
193          --send START ADDR
194         rs232_0Data <= 's';
195         rs232_0Send <= '1', '0' after 1 ns;
196         wait for 10*rs232Period;
197
198         --send DATABYTE
199         rs232_0Data <= '0';
200         rs232_0Send <= '1', '0' after 1 ns;
201         wait for 10*rs232Period;
202
203         --send DATABYTE
204         rs232_0Data <= '0';
205         rs232_0Send <= '1', '0' after 1 ns;
206         wait for 10*rs232Period;
207
208         --send DATABYTE
209         rs232_0Data <= '0';
210         rs232_0Send <= '1', '0' after 1 ns;
211         wait for 10*rs232Period;
212
213         --send DATABYTE
214         rs232_0Data <= '0';
215         rs232_0Send <= '1', '0' after 1 ns;
216         wait for 10*rs232Period;
217
218         --send DATABYTE
219         rs232_0Data <= '0';
220         rs232_0Send <= '1', '0' after 1 ns;
221         wait for 10*rs232Period;
222
223         --send DATABYTE
224         rs232_0Data <= '0';
225         rs232_0Send <= '1', '0' after 1 ns;
226         wait for 10*rs232Period;
227  ------------------------------------------------------
228
229         --send LENGTH
230         rs232_0Data <= 'l'; --length!
231         rs232_0Send <= '1', '0' after 1 ns;
232         wait for 10*rs232Period;
233
234         --send DATABYTE
235         rs232_0Data <= '0';
236         rs232_0Send <= '1', '0' after 1 ns;
237         wait for 10*rs232Period;
238
239         --send DATABYTE
```

```
240       rs232_0Data <= '0';
241       rs232_0Send <= '1', '0' after 1 ns;
242       wait for 10*rs232Period;
243
244       --send DATABYTE
245       rs232_0Data <= '0';
246       rs232_0Send <= '1', '0' after 1 ns;
247       wait for 10*rs232Period;
248
249       --send DATABYTE
250       rs232_0Data <= '5';
251       rs232_0Send <= '1', '0' after 1 ns;
252       wait for 10*rs232Period;
253
254 ------------------------------------------------------
255
256       --send READ
257       rs232_0Data <= 'r'; --read!
258       rs232_0Send <= '1', '0' after 1 ns;
259       wait for 10*rs232Period;
260
261 ------------------------------------------------------
262
263
264 --      --send DATABYTE
265 --      rs232_1Data <= "00010001";
266 --      rs232_1Send <= '1', '0' after 1 ns;
267 --      wait for 10*rs232Period;
268 --
269 --      --send DATABYTE
270 --      rs232_1Data <= "00010010";
271 --      rs232_1Send <= '1', '0' after 1 ns;
272 --      wait for 10*rs232Period;
273 --
274 --      --send DATABYTE
275 --      rs232_1Data <= "00010011";
276 --      rs232_1Send <= '1', '0' after 1 ns;
277 --      wait for 10*rs232Period;
278 --
279 --      --send DATABYTE
280 --      rs232_1Data <= "00010100";
281 --      rs232_1Send <= '1', '0' after 1 ns;
282 --      wait for 10*rs232Period;
283 --
284 --      --send CONTROLBYTE
285 --      rs232_1Data <= "11001011";
286 --      rs232_1Send <= '1', '0' after 1 ns;
287 --      wait for 10*rs232Period;
288 --
289 --      --send OUTPUTBYTE
290 --      rs232_1Data <= "10100000";
291 --      rs232_1Send <= '1', '0' after 1 ns;
292 --      wait for 10*rs232Period;
293 --
294 --      --send DATABYTE
295 --      rs232_1Data <= "00010001";
296 --      rs232_1Send <= '1', '0' after 1 ns;
297 --      wait for 1*rs232Period;
298
299     --end if;
```

```
300
301
302
303 --    rs232_0Send <= '0';
304 --    rs232_1Send <= '0';
305 --    wait for 4*rs232Period;
306 --
307 --    rs232_0Data <= 'h';
308 --    rs232_0Send <= '1', '0' after 1 ns;
309 --    wait for 15*rs232Period;
310 --
311 --    rs232_0Data <= 'e';
312 --    rs232_0Send <= '1', '0' after 1 ns;
313 --    wait for 15*rs232Period;
314 --
315 --    rs232_0Data <= 'l';
316 --    rs232_0Send <= '1', '0' after 1 ns;
317 --    wait for 15*rs232Period;
318 --
319 --    rs232_0Data <= 'l';
320 --    rs232_0Send <= '1', '0' after 1 ns;
321 --    wait for 15*rs232Period;
322 --
323 --    rs232_0Data <= 'o';
324 --    rs232_0Send <= '1', '0' after 1 ns;
325 --    wait for 1*rs232Period;
326
327 --    rs232_1Data <= 'b';
328 --    rs232_1Send <= '1', '0' after 1 ns;
329 --    wait for 10*rs232Period;
330 --
331 --    rs232_1Data <= '0';
332 --    rs232_1Send <= '1', '0' after 1 ns;
333 --    wait for 10*rs232Period;
334 --
335 --    rs232_1Data <= 'y';
336 --    rs232_1Send <= '1', '0' after 1 ns;
337 --    wait for 1*rs232Period;
338
339
340
341     wait;
342   end process;
343
344
345   process
346     variable txData: unsigned(7 downto 0);
347   begin
348     rxd0 <= '1';
349
350     wait until rising_edge(rs232_0Send);
351     txData := to_unsigned(character'pos(rs232_0Data), txData'length);
352
353     rxd0 <= '0';
354     wait for rs232Period;
355
356     for index in txData'reverse_range loop
357       rxd0 <= txData(index);
358       wait for rs232Period;
359     end loop;
```

```
360
361   end process;
362
363
364   process
365     variable txData: unsigned(7 downto 0);
366   begin
367     rxd1 <= '1';
368
369     wait until rising_edge(rs232_1Send);
370     txData := to_unsigned(character'pos(rs232_1Data), txData'length);
371
372     rxd1 <= '0';
373     wait for rs232Period;
374
375     for index in txData'reverse_range loop
376       rxd1 <= txData(index);
377       wait for rs232Period;
378     end loop;
379
380   end process;
381
382
383 --  process
384 --    variable txData: unsigned(7 downto 0);
385 --  begin
386 --    rxd1 <= '1';
387 --
388 --    wait until rising_edge(rs232_1Send);
389 --    txData := rs232_1Data;
390 --
391 --    rxd1 <= '0';
392 --    wait for rs232Period;
393 --
394 --    for index in txData'reverse_range loop
395 --      rxd1 <= txData(index);
396 --      wait for rs232Period;
397 --    end loop;
398 --
399 --  end process;
400
401
402
403 END ARCHITECTURE Flash_Prob_tester;
404
```